POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Matematica

Stochastic Optimization Assignment

GRASP solver for an integrated healthcare scheduling problem

Baitan Cristina 343428
Bergamini Cecilia 343341
Chiera Letizia 343342
Ciarfaglia Rossana 345926

Anno Accademico 2024-2025

Contents

1 The problem

Full description of the optimization problem can be found in:
https://ihtc2024.github.io/

1.1 Task Description

The optimization problem under consideration addresses three key challenges in healthcare: surgical case planning, scheduling of patient admission and assigning nurses to rooms. In particular, this problem requires the following decisions:

1. the admission date for each patient
2. the room for each admitted patient for the duration of their stay
3. the nurse for each room during each shift of the scheduling period
4. the operating theater (OT) for each admitted patient

1.2 Basic concepts

We create seven classes to memorize all the informations about physical and human resources involved in the problem:

- class Hospital :
  we memorize the number of room, OT, surgeon, nurse.
- class Nurse :
  it contains the id, the skill level and the working shifts.
- class Occupant:
  it contains the id, the gender, the age group, the length of stay, the workload produced, the skill level required and the room id.
- class OperatingTheater:
  the attributes are the id and the availability.
- class Patient:
  these are all the attributes of the class: id, if the patient is mandatory, gender, age, length of stay, surgery release day and surgery due day, surgery duration, surgeon id, incompatible room ids, workload produced and skill level required.
- class Room:
  it contains the id, the capacity.
- class Surgeon:
  it contains id and the list of the maximum surgery time for each day.

1.3 Decision variables

We define decision variables as follow:

- Decision for admission of each patient

  $$\vec{v} \in \{0,1\}^{\text{len(patients)}}$$

  where $\vec{v}[i] = 0$ indicates that patient i is not admitted, while $\vec{v}[i] = 1$ otherwise.

- The admission date for patients

  $$\vec{x} = [T^a_1, T^a_2, \ldots, T^a_{len(patients)}] \in \{0,1,2,...D\}^{len(patients)}$$

  where T^a_i = admission date patient i

- The allocation of a room for each admitted patient

  $$\vec{y} = [IdS_1, IdS_2, \ldots, IdS_{len(patients)}] \in \{0,1,2,...len(rooms)\}^{len(patients)}$$

  where IdS_i = id of the room where the patient i is located.

- The assignment of a single nurse to each room, for each shift and each day

  $$[\vec{L}]_{ijk} = \text{id nurse working in room i in day j and shift k}$$

  where $i = id_{room}$, $j = day$, $k = shift$. So $\vec{L}$ is a 3-dimensional matrix.

- The assignment of OTs to patient, for each day of the scheduling period

  $$\vec{k} = [idOT_1, idOT_2, \ldots, idOT_{len(patients)}] \in \{0,1,2,...len(OTs)\}^{len(patients)}$$

  where $idOT_i$ = id of OT assigned to patient i

We store the previous decision variables in the following *numpy*'s structures:

- *adm_yes_or_no* is a binary vector that represents $\vec{v}$.
- *adm_date* is a vector that represents $\vec{x}$.
- *room_x_patient* is a vector that represents $\vec{y}$.
- *nurse_x_room* is a tensor that rapresents $\vec{L}$.
- *ot_x_patient* is a vector that rapresents $\vec{k}$.

1.4 Constraints

1.4.1 Hard constraints

- H1 : No gender mix

$$\Big(\sum_{p:\ \textbf{gender(p)=A}} \vec{p}_{ps} + \sum_{o:\ \textbf{gender(o)=A}} \vec{o}_{os}\Big)\Big(\sum_{p:\ \textbf{gender(p)=B}} \vec{p}_{ps} + \sum_{o:\ \textbf{gender(o)=B}} \vec{o}_{os}\Big) = 0 \quad \forall s$$

  where $\vec{o}_{os}$ is the same variable of $\vec{p}_{ps}$, but it refers to the occupants

- H2: Compatible rooms.

  It is satisfied by the construction of the vector $\vec{y}_{ps}$, where each patient is assigned to one compatible room.

- H3: Surgeon overtime.

$$\sum_{p \in A} \text{surgery duration}_p \leq \text{max surgery time}_{ct} \quad \forall t, \forall c$$

  where $\mathcal{A} = \{p \in P : \vec{x}_p = t \ \wedge \ \text{surgeonId}_p = c\}$,

- H4: OT overtime.

$$\sum_{p \in \tilde{P}} \text{surgery duration}_p \leq \text{availability}_{tz} \quad \forall t, \forall z$$

  where $\tilde{P} = \{p \in P : \vec{x}_p = t \ \wedge \ \text{idOT}_p = z\}$,

- H5: Mandatory versus optional patients.

$$1 \leq \vec{x}_p \leq D \quad \forall p \in \mathcal{L}$$

  where $\mathcal{L} = \{p \in P : \text{p is mandatory}\}$,

- H6: Admission day.

$$ReleaseDate_p \leq \vec{x}_p \leq DueDate_p \quad \forall p \in \mathcal{L}$$
$$ReleaseDate_p \leq \vec{x}_p \quad \forall p \in \mathcal{L}^c$$

  ,

- H7: Room capacity.

$$\sum_{p \in P'_s} \vec{p}_{ps} + \sum_{o \in O'_s} \vec{o}_{os} \leq capacity_s \quad \forall t \ \forall s$$

  where $P'_s = \{p \in P : p\text{occupies room } s \text{ at time } t\}$ and alike $O'_s = \{o \in O : o\text{occupies room } s \text{ at time } t\}$

1.4.2 Soft constraints

The soft constraints are constraints that are not strictly enforced but rather penalized when violated. To handle them, it is necessary to modify the objective function by adding a penalty term that increases as the constraint violation increases. Therefore the objective function consists in the sum of these constraints' penalities. Each soft constraint is associated with a weight w_i.

- S1: Age groups
$$\sum_s \sum_t w_{age} \cdot max_{p,p' \in P'_s \vee \in O'_s} |Age\ p - Age\ p'|$$

- S2: Minimum skill level
$$\sum_n \sum_{p \in P,O} \sum_{t=\vec{x_p}}^{min(\text{length of stay},D)} w_{skill} \cdot [\ \text{level } \vec{\text{request}}_{pt} - \vec{\text{skill}}_{nurse\ t}]_+$$
where level request$_{pt}$ is the vector that containes the skill level required for each shift by patient p on day t.

- S3: Continuity of care
For these constraints, a penalty is applied when the number of nurses assigned to a single patient exceeds 3 (one nurse per shift). Obviously, the penalty increases as the number of nurses increases.
$$\sum_p w_{continuity} \cdot (\sum_n \beta_{np})$$
where β_{np} is 1 if the patient p is assigned to the nurse n at least one time.

- S4: Maximum workload for nurse
$$\sum_n \left(\sum_{t=1}^{D} w_{workload} \cdot \left(\sum_{\substack{p,o \\ \text{s.t. } p,o \text{ associated to } n}} \text{workload produced}_{p\ 3t-3} \right) - \text{max load}_{n;3t-3} \right)+$$
$$\sum_n \left(\sum_{t=1}^{D} w_{workload} \cdot \left(\sum_{\substack{p,o \\ \text{s.t. } p,o \text{ associated to } n}} \text{workload produced}_{p\ 3t-2} \right) - \text{max load}_{n;3t-2} \right)+$$
$$\sum_n \left(\sum_{t=1}^{D} w_{workload} \cdot \left(\sum_{\substack{p,o \\ \text{s.t. } p,o \text{ associated to } n}} \text{workload produced}_{p\ 3t-1} \right) - \text{max load}_{n;3t-1} \right)$$
where $3t-3, 3t-2, 3t-1$ represent the 3 shifts on the day t, workload produced$_{p\ i}$ represents the workload request of patient p for shift i and finally max load$_{n;i}$ is the maximum workload of the nurse n for shift i.

- S5: Open OTs
$$\sum_t \alpha_t \cdot w_{open}$$
where α_t indicates the number of open operating teatres at time t

- S6: surgeon transfer

$$\sum_{c} \sum_{t} \gamma_{ct} \cdot w_{change}$$

where γ_{ct} is the number of different OTs of the patient associated with the surgeon c at time t

- S7: Admission delay

$$\sum_{p \in \hat{P}} w_{delay} \cdot (\vec{x}_p - \text{release day}_p)$$

where $\hat{P} = \{p \in P : \vec{x}_p \leq D\}$

- S8: Unscheduled patients

$$\sum_{p} w_{\text{not accepted}} \cdot (1 - \vec{v}_p)$$

2 GRASP

The GRASP method will be utilized to address this constrained non-differentiable optimization problem.
The Greedy Randomized Adaptive Search Procedure (GRASP) is an optimization method that repeatedly applies local search from different starting feasible solutions to find good quality solutions, though not necessarily the global minimum. At each step of the local search, the neighborhood $N(x)$ of the current solution x is explored to find a solution $x' \in N(x)$ such that $f(x') < f(x)$. If such an improving solution is found, it becomes the new current solution, and another local search step is performed, i.e., $x \leftarrow x'$. Additionally, if $f(x') < f(x^*)$, the best-known solution x^* is updated accordingly. If no improving solution is found, the procedure selects a new starting point either by restarting from scratch or by applying a random modification to the current solution. In our version, we opt to build the solution from scratch in a random way.

2.1 Function *grasp_solver*

This function aims at apply Grasp method to the Integrated Healthcare Timetabling problem returning a solution which should be both feasible and it should minimize the objective function.
In this function first of all we randomly create a feasible random point through the function *construct_feasible_solution*. This function returns a solution $x\_feasible$ and $flag_point_found$1: True if a feasible solution is found, False otherwise.
If no feasible solution is found, the function *construct_feasible_solution* is recalled, this time we relax the constraints of the optimization problem by not admitting any non-mandatory patient by default. If a feasible solution is still not found after a maximum number of iterations, a warning is raised. Through the function *evaluate_obj_func* , we assess the quality of the solution, as the goal is to store the best solution found so far.
The algorithm proceeds with a search in the surroundings: it is called the function *LocalSearch* which explores the neighborhood of the current solution to find improvements. If an improved solution is found, the best solution is updated and a new local search is done nearby this new point.
Below is the pseudo-code version of *grasp_solver* function:

Algorithm 1 GRASP Solver

Input: D, $weights$, $occupants$, $patients$, $operating\_theaters$, $rooms$, $nurses$, $surgeons$, max_iter
Output: Best solution x_{best} and its value f_{best}
Initialize:
$iter \leftarrow 1$,
$flag\_point\_found2 \leftarrow$ `True`
while $iter < max\_iter$ **and** $flag\_point\_found2$ **do**
    Construct feasible solution:
        $(x_{\text{feasible}}, flag\_point\_found1) \leftarrow$ ConstructFeasibleSolution$(\dots)$
    if not $flag\_point\_found1$ **then**
        Relax non-mandatory admission:
            $(x_{\text{feasible}}, flag\_point\_found2) \leftarrow$ ConstructFeasibleSolution$(\dots,$ `relax=True`$)$
        if not $flag\_point\_found2$ **then**
            return infeasible solution
        end if
    end if
    $f \leftarrow$ EvaluateObjective(x_{feasible})
    if $iter = 1$ **or** $f < f_{\text{best}}$ **then**
        $f_{\text{best}} \leftarrow f$,
        $x_{\text{best}} \leftarrow x_{\text{feasible}}$
    end if
    $improvements \leftarrow$ `True`
    while $improvements$ **do**
        $(x_{\text{neigh}}, improvements, f_{\text{new}}) \leftarrow$ LocalSearch(x_{feasible})
        if $improvements$ **then**
            $f_{\text{best}} \leftarrow f_{\text{new}}$,
            $x_{\text{best}} \leftarrow x_{\text{neigh}}$,
            $x_{\text{feasible}} \leftarrow x_{\text{neigh}}$
        end if
    end while
    $iter \leftarrow iter + 1$
end while
return $(f_{\text{best}}, x_{\text{best}})$

2.2 Function *construct_feasible_solution*

This function aims to construct a feasible solution that satisfies all hard constraints related to room assignments, operating theaters, nurse scheduling, and patient admission dates. The process iterates up to a maximum of $max\_iter = 200$ times to identify a valid configuration.
The first step involves the function *admit_constr* , which builds feasible solutions by addressing the hard constraints H5, H6, H3, and H4. These ensure that patient admissions are compatible with the admission constraints and availability of operating theaters and surgeons.
Next, the solution is refined using the *room_constr* function, which adjusts patient allocations to satisfy constraints H1, H2, and H7 related to room occupancy and compatibility.
Subsequently, the $follow\_shift$ function assigns nurses to specific room shifts over a series of days, ensuring that each nurse is scheduled only for shifts they are available to work.
Before diving into some details of the previously mentioned functions, it is important to underline the line of thought that led us to create them. Since many of the constraints are dependent on the admission dates, we begin by using *admit_constr* to assign feasible admission dates. Only afterward we assign patients to rooms through the *room_constr* function.

2.2.1 *admit_constr*

This function aims at modifying the vector of decision variables *adm_yes_or_no*, *adm_date* and *ot_x_patient* in order to satisfy constraints regarding the admission date, surgeon and OT. The function first processes the mandatory patients given their priority.
Each mandatory patient is processed in a loop:

- The admission date is randomly chosen within the intersection of the surgeon's availability and the patient's range of admission (from release date to surgeon date). Furthermore the admission date is chosen in order to avoid breaking the constraint regarding surgeon capacity.
- The OT is randomly chosen from OTs with enough capacity.

If no solution is found, the function returns a boolean value indicating that the admission procedure should be retried. To increase the chances of finding a solution, we introduce randomness by shuffling the patients before processing them, allowing a broader exploration of possible outcomes.

After doing this, we handle the non-mandatory patients: similar to mandatory patients, the algorithm tries to satisfy both the surgeon's and operating theater's constraints. If no valid date or no valid operating theater is found the patient will not be admitted.

2.2.2 *room_constr*

The function ensures that the capacity limit of each room is not exceeded (H7), that all individuals in a room share the same gender (H1) and that no patient is assigned to incompatible rooms (H2). We assign rooms to admitted patients, ensuring that each room hosts individuals of the same gender throughout the patient's stay, while also respecting the room's capacity constraints.
Before doing this, we shuffle occupants and patients in order to introduce randomness.
This function populates the data structure *room_x_patient*.

2.2.3 $follow\_shift$

This function populates the data structure $nurse\_x\_room$ meeting nurse's hard constraints.

2.3 Function $evaluate\_obj\_func$

This function evaluates the objective function at a given point, specifically by deriving and summing the penalties regarding soft constraints.

- S1: $qnt1$ considers the age differences in the rooms.
- S2-S3: $qnt2$ measures the mismatch between the required and provided nurse skill levels, instead $qnt3$ measures nurse continuity.
- S4: $qnt4$ is a variable that accumulates the penalty related to nurse workload overload.
- S5: $qnt5$ tracks the number of operating theaters used each day and adds a penalty to the objective function for each open OT.
- S6: $qnt6$ memorizes the number of different rooms a surgeon is assigned to in a single day.

All the quantities are then multiplied by the weights to represent the cost of penalties. The total objective function cost is the sum of all constraint penalties.

2.4 Function $LocalSearch$

The function implements the local search which tries to improve an existing solution through a series of perturbations to find a better solution iteratively.
perturbations is a list of possible perturbations those represent different types of changes that will be applied to the current solution. The list is shuffled and the function applies each perturbation sequentially till a better solution is found. The shuffle is useful to change the order of perturbation every time and so enhance exploration.
The perturbations are

- "$admission\_forward$": it changes the admission date for a selected subset of patients by randomly selecting a new date, advancing it by either 1 or 2 days.
- "$admission\_backward$" The admission date is moved backward by either 1 or 2 days.
- "$room\_change$". In this perturbation, the room assignment for a patient is changed to a different room (excluding incompatible rooms).
- "$nurse\_swap$" This perturbation shuffles nurse assignments within the same shift and day. Specifically, it swaps the rooms assigned to nurses who are already scheduled in the same shift, ensuring that no hard constraints are violated. Notably, this is the only perturbation that guarantees hard constraint feasibility.

After each perturbation, the resulting solution is checked for feasibility using the function *check_constraint*. Following the first three perturbations ($admission\_forward$, $admission\_backward$, and $room\_change$) and their corresponding feasibility checks, we apply the fourth perturbation, $nurse\_swap$, multiple

times. Since this operation preserves feasibility (it does not violate hard constraints), it is computationally inexpensive and does not require additional checks.
The local search strategy adopted is the First Improvement approach: the current solution is immediately updated as soon as a feasible improvement is found. This choice was made to minimize the computational associated with repeatedly verifying the feasibility of candidate solutions.

2.5 Function *check_constraint*

Checks whether the given solution satisfies all the hard constraints. It returns True if all hard constraints are satisfied, and False otherwise.
This function evaluates a set of constraints related to room assignments, operating theater usage, surgeon availability, patient admission schedules, and other hospital-specific constraints through the following functions: *room_constr_bool*, *OT_and_Surgeon_constr_bool*, *bool_period_of_admission_constr*, *bool_admit_mandatory_constr*, *bool_incompatible_room_constr*. Each of these functions returns a boolean value: True if the associated constraints are satisfied, False otherwise.

2.5.1 *room_constr_bool*

This method checks the constraints H1 and H7: verifies whether the room assignments for patients comply with the room capacity constraint and the gender constraint.
Return *False* if any capacity or gender constraint is violated.

2.5.2 *OT_and_Surgeon_constr_bool*

This function verifies constraints H3, H4: surgeons' total surgery time on each day does not exceed their maximum allowed time and operating theaters' total surgery time on each day does not exceed their available time.

2.5.3 *bool_period_of_admission_constr*

It evaluates constraint H6 so it checks if the admission dates for the admitted patients meet constraints regarding the surgery release day and, only for mandatory patients, the surgery due day.

2.5.4 *bool_admit_mandatory_constr*

This method performs a check on constraint H5.
Checks if all mandatory patients have been admitted. If any mandatory patient is not admitted, it returns *False*.

2.5.5 *bool_incompatible_room_constr*

It performs a check on constraint H2.
Verifies that all patients are assigned to compatible rooms. Specifically, it checks if a patient is assigned to a room listed in their incompatible room IDs.

3 Results

3.1 Trajectory of objective function through iteration

Below are four images showing the objective function value as the number of iterations increases for four of the proposed files: *i03*, *i04*, *i07*, *i11*.

Legend:

- **Light blue points:** objective function values evaluated at the points found during the local search, which explores the neighborhood of the current random point to identify potential improvements. Only points which improves the best solution so far are plotted.
- **Orange points:** objective function values evaluated when the method restarts from scratch, picking randomly a new feasible state.

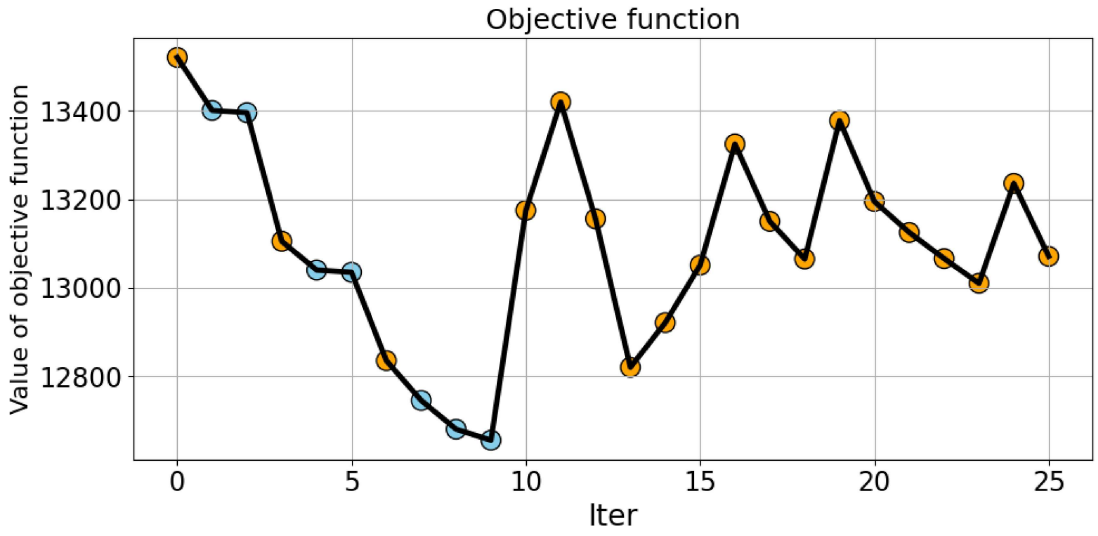

Figure 1: Objective function i03

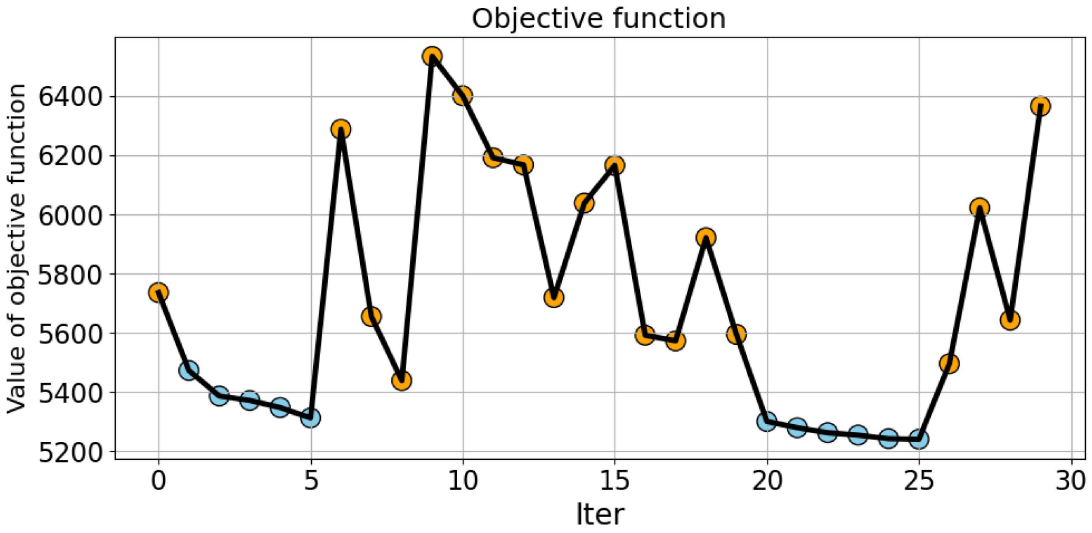

Figure 2: Objective function i04

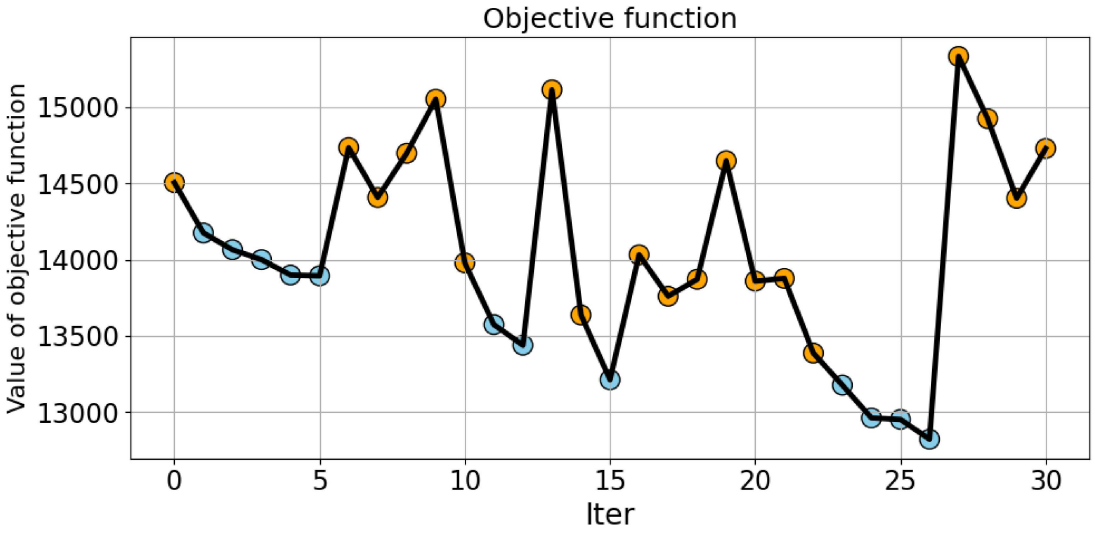

Figure 3: Objective function i07

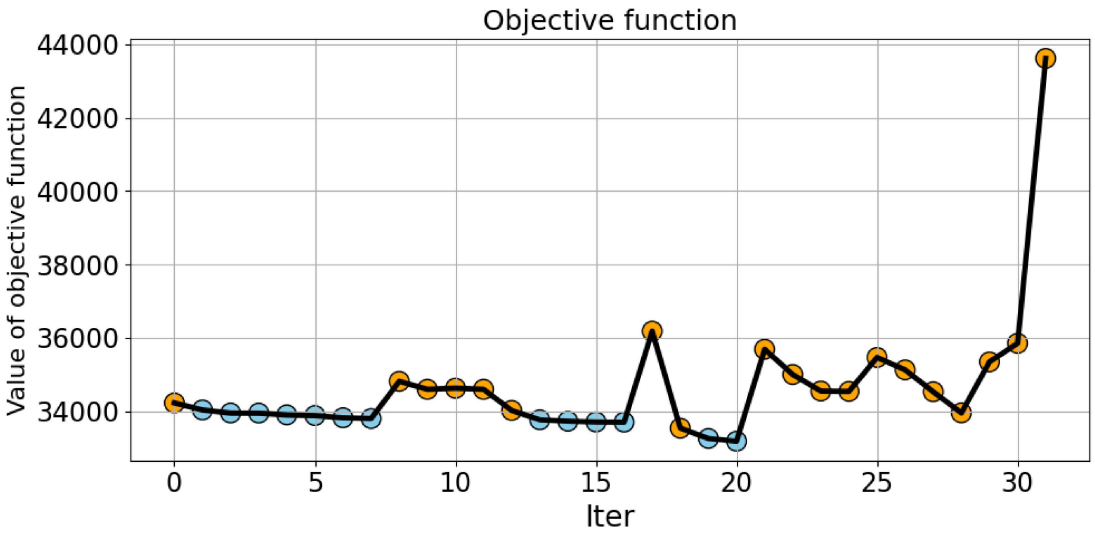

Figure 4: Objective function i11

As we can seen from the previous image objective functions generally decrease as the number of iterations increases, although there are multiple peaks and drops throughout the process due to the randomness of the starting solution.

3.2 Comments on solution and computational costs

i03	i04	i07	i11
12290.0	5220.0	12642.0	33139.0
38.75 s (93.42%)	258.05 s (86.16%)	304.09 s (95.74%)	159.57 s (95.00%)
2.73 s (6.58%)	14.46 s (13.84%)	13.52 s (4.26%)	8.40 s (5.00%)
20.74 s	136.26 s	158.81 s	83.81 s

Table 1: Table summarizing three different types of data for the four documents

In each row the table contains:

- First row: the objective function value of the final result.
- Second row: mean time for creating the initial random point and, in parentheses, the percentage of the total time.
- Third row: mean time for searching for a better solution in the neighborhood and, in parentheses, percentage of the total time.
- Fourth row: mean time to find a result.

Comments:
It is evident that the computational cost, in terms of time, to generate a new feasible random solution is significantly higher than the one required to explore the neighborhood.
Nevertheless, the average time to construct a feasible solution varies considerably between the instances:

- For *i07*, the construction is the most time-consuming, requiring on average 304.09 seconds and approximately 867 iterations.
- In contrast, *i03* requires only around 230 iterations to build a feasible solution, making it significantly faster.

To better understand why it was difficult to find a feasible solution, especially which hard constraints were the most problematic, we grouped the constraints into two categories. *Group 1* includes $H3$, $H4$, $H5$, and $H6$ (related to operating theaters, surgeons, and admission dates), and *Group 2* includes $H1$, $H2$, and $H7$ (related to room assignments). We calculated the average number of violations for each group during the iterations and found the following:
In *i07*:

- Group 1 has an average of 28 violations.
- Group 2 is considerably more problematic, with an average of 45 violations.

In *i03*:

- Group 1 exhibits an average of 17 violations.
- Group 2 has an average of only 0.16 violations, which is practically negligible.

This indicates that $i03$ almost never violates constraints related to Group 2 ($H1$, $H2$, and $H7$), whereas in file $i07$, violations of these constraints are significant.
The performance differences between the two instances could be related to the number of occupants and mandatory patients. Indeed, the notable difference between the two instances is that the number of occupants and mandatory patients is lower in file $i03$. Therefore, it seems that a higher number of these two can increase the difficulty to find feasible solution.

In the columns of the following table we report:

- First column: number of patients
- Second column: number and percentage of mandatory patients
- Third column: number of occupants
- Fourth column: number of available operating theaters
- Fifth column: number of available rooms
- Sixth column: number of nurses working during all the period
- Seventh column: number of surgeon working during all the period

	patients	mandatory	occupants	OT	rooms	nurses	surgeon
i03	45	7 (15.6%)	3	2	4	28	1
i07	79	25 (31.7%)	20	4	13	28	3

Table 2: Table summarizing the number of patients, occupants, human and non human resources for the files $i03$ and $i07$

3.3 Comments on soft constraints violated

	S1	S2	S3	S4	S5	S6	S7	S8
i03	13	154	20	91	0	0	13	24
i04	58	453	55	196	6	7	35	1
i07	59	620	84	271	11	23	51	3
i11	20	445	63	313	0	12	32	61

Table 3: Table summarizing three different types of soft constraints for the four documents

Each entry in the table indicates the number of constraint violations.
Rows represent the files analyzed, while columns correspond to the constraints considered:

- first column:
  It counts how many times each rooms contains patients from different age groups
- second column:
  It indicates how many times a mismatch in nurse skills occurs.
- third column:
  It refers to the continuity of care. We consider a violation if the number of nurses assigned to a patient or occupant exceed 3 during their stay.
- fourth column:
  It regards nurses' excessive workload: we count how many times nurse exceeds their maximum workload.
- fifth column:
  If the number of operating theaters exceeds one, there is a violation.
- sixth column:
  We count the surgeon transfers across operating theaters. Each transfer is considered a violation.
- seventh column:
  The number of times a delay between a patient's surgery release day and their admission day occurs.
- eighth column:
  The number of unscheduled non-mandatory patients.

Analyzing the table, we can draw the following conclusions regarding constraint violations and their associated costs:

- **Least Violated Constraints**:
  - For files *i03* and *i11*, the least violated constraint is **Constraint 5**, which concerns the number of operating theaters exceeding the limit of one. These files show no violations of this constraint.

- For files *i04* and *i07*, the least violated constraint is **Constraint 8**, which concerns the number of unscheduled non-mandatory patients.

- **Most Violated Constraint**:
  - Across all files, **Constraint 2** is the most frequently violated. In particular, file *i07* shows 620 violations of this constraint.
    Likely because it has a relatively low penalty weight:
    i03: 5, *i04*: 1, *i07*: 10, *i11*: 1.

- **Highest Penalty Weights**:
  - The most expensive constraint in terms of penalty weight is the **Constraint 8**, with the following values:
    i03: 400, *i04*: 300, *i07*: 450, *i11*: 500.
  - The second most expensive constraint is **Constraint 5**, with weights:
    i03: 50, *i04*: 20, *i07*: 20, *i11*: 30.
    As expected, due to their high penalty weights, these constraints are among the least violated. Specifically, as already highlighted, in file *i03* and *i11*, the least violated constraint is Constraint 5, while in *i04* and *i07*, it is Constraint 8.

3.4 Design Choices and Considerations

- We initially considered discarding all low-quality solutions and skipping local search for them. However, this approach presented several issues. First, we were uncertain about how to define an appropriate threshold for rejection. Second, we lacked knowledge of the objective function's behavior in the vicinity of poor solutions; it might increase, remain flat, or even improve unpredictably. Given these uncertainties, we ultimately decided to apply local search to all feasible solutions, regardless of their initial quality.

- Initially, we used a uniform probability to decide whether to admit non-mandatory patients. However, we later discovered that setting this probability to one, thus admitting all non-mandatory patients at the start, led to a decrease in the objective function, since the penalty for rejecting non-mandatory patients is consistently high across all input files. As a result, we chose to admit all patients initially, and only reject non-mandatory ones later if necessary to satisfy other constraints.

- Initially, we imposed a constant room gender constraint for the entire duration of a patient's stay. Later, realizing that this requirement was not explicitly specified, we relaxed the constraint, which resulted in improved solution quality.

- Some factors in our approach were fixed based on the characteristics of the input data, selecting values that offered a good trade-off between solution quality and computational cost. For instance, in the local search phase, the number of perturbations was set to a fixed value. However, from a modeling perspective, such parameters could be treated as hyperparameters to be tuned. Increasing the number of allowed perturbations, for example, could lead to a more thorough search and better exploitation reflecting the classic trade-off between exploration and exploitation.