POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Matematica

Report Stochastic Optimization

GRASP

Chiodo Martina - 343310
Vigè Sophie - 339268

Anno Accademico 2024-2025

1 Data structures

The problem considered is the scheduling of surgeries and patients' admission in a single healthcare system. We are provided with different entities populating the system, and for each of them we created a corresponding class:

- Hospital: even if there is only one hospital considered, we use this object to store and update information about the rooms (their capacity, availability at a certain time, sex of the people in it) and the operating theaters (their availability). Here there is also a function (`creating_matrix_dayxroomxpatients`) that creates a 3-dimensional matrix that will be useful to verify constraints: it is a list of length the number of days containing, for each day, a list of length the total number of rooms that contains a list of all the people staying in the said room during the said day.
- Nurse: nurses are identified by their id and the shifts they are working are given and fixed.
- Surgeon: they are treated similarly to the nurses, but their maximum surgery time on available days must be respected (it is an hard constraint, differently from the nurses' one).
- Occupant: patients that are already in the hospital at the beginning of the considered period of time.
- Patient: patients still to be admitted. Their admission can either be mandatory or not.
- Problem: we store here all the dictionaries containing the istances of the previous classes. In this way we can verify the constraints and generate feasible states.
- State: a certain schedule for the hospital; some structures are used to memorize it. First, the scheduling of patients' admissions. It is stored in a dictionary (`dict_admission`) which has as key the id of the patient and as a value a vector of three elements: the operating theatre his surgery will take place in, the date of the surgery (which coincides with the admission date and it is setted to -1 if the non mandatory patient is not admitted within the considered scheduling period) and the room the patient is staying in. Then, a dictionary (`nurses_shifts`) that has as key nurses' id and as value a vector (with length the total number of shifts in the period of time) containing -1 if the nurse is not working and otherwise the id of the rooms he is assigned to (they can be even more than one per shift). Lastly the scheduling for the operating theaters: a dictionary (`scheduling_OTs`) with key the id of the OT and as value a vector (with length the total number of days) that for each day contains the list of ids of the patients that undergo surgery in that OT that day. We store here also the 3-dimensional matrix introduced above, when created using the method of the Hospital class.

2 Hard and soft constraints

Hard constraints must be satisfied for a state to be feasible, while soft constraints if not satisfied produce a penalty in the objective function. The costs of these penalties are given and fixed.

In order to verify the feasibility of a given state we start by checking that all the operating theaters, rooms and admission dates exist.

Then, for each nurse we check that they have at least a room assigned for the shifts they are supposed to be working and no room assigned during the shifts they are unavailable.

For every room, we check that it is not incompatible for the people staying in it, that there is not gender mix among the patients and that the maximum capacity of the room is respected. We also need to check that there is always at least a nurse assigned to it (if the room is not empty).

For every mandatory patient we check that they are admitted within the scheduling period and, more specifically, their admission date should be between their release date and their due date. For non-mandatory patients that are admitted, the check is just on the release date, since they do not have a due date.

Lastly, for every surgeon and operating theatre, we check that the total amount of time of the scheduled surgeries does not exceed the maximum available for every considered day.

For soft constraints, we penalize in the eveluation of the objective function the following: admission delays (we want to minimize the number of days between a patient's release date and their admission) and unscheduled non-mandatory patients, opening of operating theaters, surgeon transfers. For nurses, the eventual excess of

their maximum workload, the continuity of care (we try to minimize the number of different nurses that take care of a patient) and the minimum skill required for a patient, that might not be respected by the nurse.

3 Neighborhoods generation

The *Greedy Randomized Adaptive Search Procedure* (GRASP) is based on repeated local search in the neighborhood of a given point. We generate, for each given state, a set of states obtained by perturbation of the given one. When doing so, we try to check some of the hard constraints in order not to generate too many neighbors that are not feasible states. The perturbations performed are the following:

- If a non-mandatory patient was not admitted within the scheduling period, we admit him. We generate several new states, admitting the considered patient in every possible date after his release date and in every possible room.
- If a patient is mandatory, we generate new states by adding or subtracting a day to their admission date (if this makes sense, for example we cannot add +1 to the admission date of a patient that was admitted on the last day of the scheduling period).
- If a non-mandatory patient was admitted, we generate a new state where we do not admit him within the scheduling period.
- For every admitted patient, we perturbate the room they are assigned to, by adding or subtracting 1 to the id of the room (or just adding or subtracting if they are in the first or last room).
- For every admitted patient, we change the operating theater they are assigned to by choosing randomly between the available ones.
- For all the nurses, we generate some new states where we exchange the rooms that two nurses working in the same shift are asssigned to.

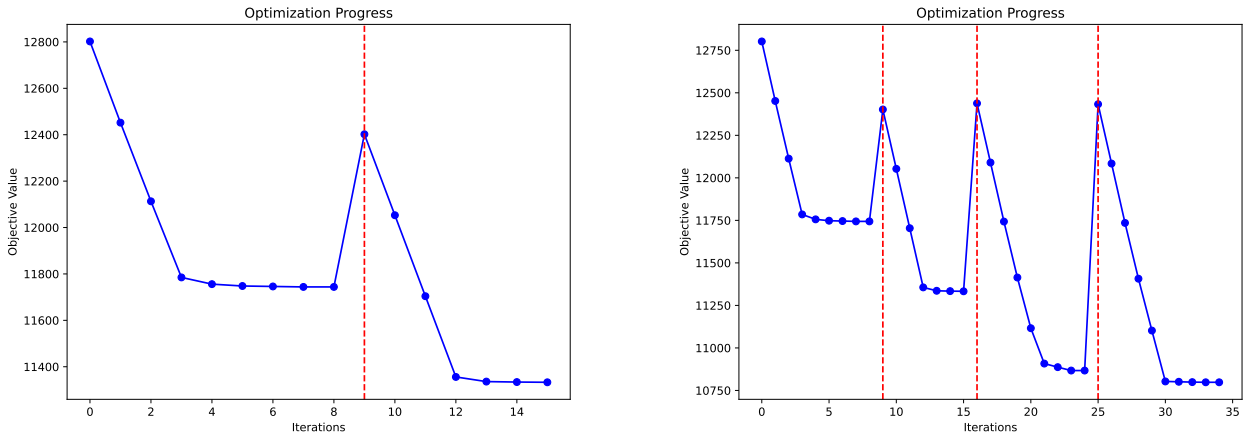
4 GRASP implementation and outcome

The idea of GRASP is the following: given an initial feasible state, we generate and explore all its neighborhood, updating the current solution with the best found in it. If such a current solution is found and it is different from the previous one, we repeat the local search step in a neighborhood of this new state. Otherwise we generate a new feasible starting state and repeat.

In order to generate randomly a feasible state we have to check that all the hard constraints are respected. If a state is generated completely randomly it will be very likely unfeasible. So, in the class `problem`, the function `generating_feasible_state` is built in such a way that it returns a feasible state that is partially random, but generated taking into account some of the hard constraints in order not to discard too many generated states and save computational time.

The admission date for each mandatory patient is chosen randomly between their release date to their due date, exluding the days their assigned surgeon is not working. The room a patient is assigned to is chosen randomly between the ones that are not already full and we consider every room to be "labeled" with the sex of the patients in it, so we exclude those belonging to the opposite sex. For the sake of simplicity we don't admit non-mandatory patients (this is not restrictive, since in the generation of neighborhood we will admit them). We assign the nurses to the rooms by counting how many nurses are working in each shifts; in this way we decide how many rooms should a nurse cover (for example if we have 9 working nurses and 9 rooms we assign one room per nurse, while if we only have 4 nurses, we give 2 rooms each to the first three and 3 to the last nurse). The room each nurse is assigned to is chosen randomly.

All the elements needed to run the GRASP on the given problem have been built. The result of a run is the following.



(a) One restart allowed

(b) Three restarts allowed

Figure 1: Cost of generated feasible solutions. The red line corresponds to restarts occurring when no improving solution is found in the neighborhood of the current point.

In the first case, the best value of the objective function found is 11466, while in the second case it is 11127.

Red lines indicates when the method restarts from scratch, picking randomly a new feasible state. That's why the value of the objective function can increase here, while it is always non-increasing in the other points, where exploration of the neighborhoods is performed.

5 Results

We have run few experiments using the datasets made available by the Integrated Healthcare Timetabling Competition. These have been run by fixing the number of restart equal to 2.

The datasets are very different from one another, also differing in the length of the timeline, consequently it is normal that the objective function assume values of different magnitude.

Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
5294	6904	10867	6921	16289
3177	1583	10184	2332	15713
117.42 sec	99.00 sec	71.01 sec	135.80 sec	117.06 sec

Table 1: Comparison between the solution found by the GRASP solver (first row) and the actual minimum value of the objective function (second row). The third row contains the time of execution.

Due to the fact that the GRASP solver belongs to the class of local optimizers, there is no guarantee of convergence to the global minimum: the algorithm can easily get stuck in local minima. Therefore, it is not surprising that the solutions found by our solver are always greater than the actual minimum value of the objective function.

In addition, the performance of the GRASP solver is largely affected by the choice of the initial guess. In our code, the initial guess is generated without the aim of improving the objective function, and thus it is possible that the algorithm starts from points very far from the minimum, making it more difficult to reach it.

The times of execution (shown in the third row of Table 1) are measured as the elapsed time from when the `solve()` function of the GRASP solver is called to when it returns the best state it has found. These times are also significantly influenced by the quality of the neighborhood being explored. If the solver enters a promising neighborhood, it iterates for a longer time before reaching a minimum; on the other hand, if the initial guess is already very close to a local minimum, the search time can be very short.