Politecnico di Torino

Master's Degree in Data Science and Engineering

Genetic Algorithm for the Integrated Healthcare Timetabling Problem (IHTP)

Name: Giuseppe Mallo
Student ID: 346884
Academic Year: 2024/2025

1 Introduction

This report presents the implementation of a genetic algorithm applied to the Integrated Healthcare Timetabling Problem (IHTP), which consists of jointly scheduling patients, operating rooms, nurses, and other hospital resources under hard constraints (e.g., compatibility, capacity, availability) and soft objectives, quantified by a cost function to be minimized.

2 Structure of the code

The code developed to implement the Genetic Algorithm is organized into classes. Specifically, the following classes represent the entities involved in the hospital setting: `Occupant`, `Patient`, `Nurse`, `Surgeon`, `OperatingTheater`, and `Room`. Each class includes its own set of attributes, derived from the information contained in the `.json` instances.

As for the Genetic Algorithm itself, two main classes have been used: `Chromosome` and `GeneticAlgorithm`. The former represents an individual (i.e., a possible solution) within the population, which may undergo mutations and changes simulating genetic evolution. The actual evolutionary process is implemented within the `GeneticAlgorithm` class. Below, we provide a detailed explanation of the internal structure of these two classes.

2.1 Class Chromosome

`Chromosome` is a Python class that represents a possible scheduling configuration of surgeons, patients, and nurses within rooms and operating theaters, in other words, a potential solution to the problem.

Using the function `random_initialize(self)`, a `Chromosome` object is initialized randomly. The goal of this function is to generate a feasible assignment of patients (both mandatory and, if applicable, non-mandatory), nurses, operating theaters, and rooms that satisfies the hard constraints defined by the problem. To achieve this, several validation checks are applied within the assignment functions for rooms, admission days, and operating theaters, in order to satisfy the hard constraints.

Additionally, the method `fix_uncovered_rooms()` is implemented to handle the only hard constraint that might remain unsatisfied during the initialization phase. This constraint, known as *UncoveredRoom*, penalizes chromosomes in which rooms occupied by patients are not covered by any nurse. Specifically, the method iterates over all rooms and, for each day on which a room is occupied by patients, it checks whether a nurse is assigned to each of the three shifts. If not, it randomly selects a nurse from those available in that shift during the uncovered shift.

If during the initialization of a `Chromosome` object one of the hard constraints is not satisfied, it is discarded and the algorithm skips to another `Chromosome` initalization.

The `compute_cost()` function is responsible for calculating the total number of hard constraint violations and the cost associated with soft constraint violations. This evaluation is performed using a validator provided here.

2.2 Class GeneticAlgorithm

This class represents the core of the algorithm. Its main method is `evolve()`, which is called to initiate the optimization process according to the Genetic Algorithm. Inside `evolve()`, several methods are invoked, as described below.

The `selection()` method precedes the genetic modification phase of the chromosomes and is used to select the top chromosomes with the lowest cost. These chromosomes will act as the parents from which the genetic traits of the offspring will be inherited.

The `crossover()` method combines two randomly selected parent chromosomes. It randomly chooses a subset of patients (whose size can be tuned by the user) to be exchanged between the two parents, and then returns two resulting child chromosomes. The exchange is performed only for patients with the same identifier who are both admitted, as all other cases are handled by the `mutation()` method, which also allows switching non-mandatory patients from non-admitted to admitted status, and viceversa. It should be noted that `crossover()` is invoked with a high probability; however, if the crossover is not performed, the two returned children will be simple copies of their parents, which helps preserve a degree of elitism in the population. This action could potentially result in the creation of chromosome duplicates, but the algorithm is designed to prevent this, thanks to the `hasChanged()` method, which assures that at least a crossover or a mutation has been executed on the child chromosome.

The `mutation()` method is used to modify the room, operating theater, or admission day of mandatory and non-mandatory patients who have been admitted. It can also decide whether or not to admit non-mandatory patients, or to unschedule those who were previously admitted. For nurses, with a certain probability, it may add or remove the rooms assigned to them.

After applying `crossover()` and `mutation()` to a child, the `fix_uncovered_rooms()` method is invoked. Finally, the `compute_cost()` function from the `Chromosome` class is used to calculate the number of hard constraint violations ("Total violations") through the validator, as well as the cost associated with soft constraint violations ("Total cost"). If the number of hard constraint violations is zero, the child chromosome is inserted into the population.

3 Anti-Stagnation strategies

A common issue in optimization methods is stagnation, which refers to a situation in which the algorithm gets stuck in a local minimum and is unable to progress toward better solutions or the global minimum. To address this problem, several techniques have been implemented that directly act on the population to help the algorithm move forward. These methods are described below, along with an explanation of where they are integrated in the process. We remark that all these methods have been implemented inside the `GeneticAlgorithm` class.

3.1 Injection strategy

The `injection()` method, defined in the `GeneticAlgorithm` class, is invoked when the number of consecutive generations in which the best solution remains unchanged (`num_times_best`, an attribute of the `GeneticAlgorithm` class) exceeds a certain threshold, defined by `self.stagnation` attribute of the class `GeneticAlgorithm`. In practice, the `injection()` method introduces a fixed number of newly generated chromosomes into the current population and returns a new population composed of the top `self.num_population` chromosomes with the lowest "Total cost". This strategy aims to refresh the genetic pool and help the algorithm discover new chromosome combinations that may lead to a new best solution.

However, this method may not always be effective, since it is possible that none of the newly introduced chromosomes will be selected for the next generation. To handle cases of prolonged stagnation, an additional method has been implemented: `enforce_injection()`. This method is triggered with a certain probability, but only when stagnation persists for a large number of generations.

Its logic involves selecting the current set of parents and replacing a small percentage of the worst among them with randomly generated chromosomes. The goal is not to introduce immediately optimal solutions, but to increase genetic diversity within the parents, thus enabling new crossover opportunities with previously unexplored configurations. So, even if the newly introduced chromosomes do not exhibit low costs, they may contribute useful genetic features that support the search for better solutions in future generations.

3.2 Mutation Probability Scheduling

In order to improve the convergence behavior of the genetic algorithm and avoid premature convergence, a mutation probability scheduler was implemented. This mechanism dynamically adjusts the mutation rate across generations through the `update_probabilities()` function. However, the rates are effectively adjusted only if the current best solution remains unchanged for a number of times (`num_times_best`) exceeding the user-specified threshold `stagnation`. For the experiments conducted on the different problem instances, the attribute `stagnation` was set to a value of 10.

This function is responsible for dynamically adapting several parameters of the genetic algorithm, in particular the probabilities of crossover, mutation, and the one related to schedule or unschedule non mandatory patients. If explicitly requested by setting the flag `reset_probabilities` to `True` or every n generations (in our case, every 50 generations), the function resets these parameters to their original values; otherwise, it progressively modifies them.

In this way, the mutation probability increases while the crossover probability slightly decreases, along with other adjustments, with the goal of introducing more diversity into the population, preventing the algorithm from getting stuck in local minima, increasing exploration of the search space when necessary (e.g., via more mutations).

4 Results

In this section, we show how the algorithm performed with the 30 instances proposed by the problem. Before showing the tables, we explain the meaning of the column names of the following tables:

- **dimension_population**: number of chromosomes in the population
- **num_selected**: number of top chromosomes selected (i.e. parents) at each era during the selection stage, which precedes the crossover and mutation phases
- **start mut. prob.**: the starting mutation probability
- **max mut. prob.**: the highest mutation probability admitted by the scheduler inside `update_probabilities()` function
- **max total eras**: maximum number of iterations (or eras) that can be executed inside `evolve()` function
- **executed eras**: actual number of iterations (or eras) that have been executed
- **time (s)**: execution time in seconds **total eras**
- **start eval.**: value obtained by applying the objective function to the best chromosome of the first generation
- **final eval.**: value obtained by applying the objective function to the best chromosome of the final generation

In each table, the lowest value found during the experiments is written in **bold**.

N.B.: The solution of each instance is attached to this report in the format `ch_0_cost.json`, where `0` denotes the number of hard-constraint violations, and `cost` represents the value of the objective function as computed by the validator regarding the soft constraints. All the solutions reported below are attached.

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	4496.39	6329	5218
30	12	0.1	0.5	500	500	7918.47	6063	**4898**
40	24	0.1	0.5	500	500	8076.75	6445	5362

Table 1: Results for instance i01.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	1174.06	3922	2804
30	12	0.1	0.5	500	500	2224.97	3850	2859
40	16	0.1	0.5	700	700	2643.33	3848	**2678**

Table 2: Results for instance i02.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
30	12	0.1	0.5	500	237	14021.34	12796	**11804**

Table 3: Results for instance i03.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	1167.33	7723	5209
30	12	0.1	0.5	500	500	1840.24	7408	5374
40	16	0.1	0.5	700	700	3042.76	7302	**4995**

Table 4: Results for instance i04.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	18325.67	19767	**17513**

Table 5: Results for instance i05.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	1945.73	19440	14463
30	12	0.1	0.5	500	500	2907.84	18477	14752
40	16	0.1	0.5	700	700	5362.66	19377	**14055**

Table 6: Results for instance i07.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
40	16	0.1	0.5	700	700	21604.43	23552	**19556**

Table 7: Results for instance i08.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	500	3911.56	25652	18631
30	12	0.1	0.5	500	500	8311.75	24035	16805
40	24	0.1	0.5	700	700	9670.30	22858	**16701**

Table 8: Results for instance i09.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	168	25204.12	36051	**30621**

Table 9: Results for instance i13.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	31	22809.05	27135	**23513**

Table 10: Results for instance i14.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	92	28801.26	39613	**31351**

Table 11: Results for instance i15.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	216	43207.18	120275	**100500**

Table 12: Results for instance i17.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	73	33120.43	102574	**97424**

Table 13: Results for instance i19.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	123	47580.21	62806	**55403**

Table 14: Results for instance i21.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	159	27720.78	28184	**25744**

Table 15: Results for instance i25.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	103	16204.32	176362	**145853**

Table 16: Results for instance i27.json

dimension_population	num_selected	start mut. prob.	max mut. prob.	max total eras	total eras executed	time (s)	start eval.	final eval.
20	8	0.1	0.5	500	39	16020.53	40124	**35471**

Table 17: Results for instance i29.json

After presenting the tables with the results obtained for the different instances, we can now proceed with the discussion. It should be noted that some instances were encountered in which the algorithm did not converge to any chromosome with zero hard constraint violations at the beginning; for this reason, the corresponding tables have not been included.

From the reported tables, it can also be observed that in some instances the algorithm was unable to reach the maximum number of total eras. This, however, is not a negative outcome, since the maximum number of total eras was set merely as a safeguard to prevent the Genetic Algorithm from running indefinitely. Furthermore, there are instances in which the algorithm requires more time, as it struggles to generate chromosomes that satisfy the hard constraints. This explains why the Genetic Algorithm was stopped earlier when tested on those specific instances.

Overall, the algorithm generally achieves consistent improvements towards solutions with lower cost. Moreover, for those instances where the algorithm did not take excessive time to reach the maximum number of total eras, multiple experiments were carried out with increasing population sizes. As the population size and the number of maximum allowed iterations grow, the algorithm is able to explore a broader solution space, thereby increasing the chances of finding better solutions. It should also be noted that the implemented anti-stagnation strategies really helped the algorithm overcome local minima encountered during the iterations, enabling the discovery of new and more promising configurations with lower cost.