UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics

Algorithmic Methods for Mathematical Models

Solving the Nurse Scheduling Problem using Integer Linear Programming and Meta-heuristics

Authors:
Adrián Rodríguez Bazaga
Pau Rodríguez Esmerats

Professors:
Albert Oliveras Llunel
Marc Ruiz Ramírez

January 9, 2018

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Document Structure . . . 3

2 Integer Linear Programming **4**
- 2.1 Decision variables . . . 4
- 2.2 Instance parameters . . . 4
- 2.3 Objective function . . . 4
- 2.4 Constraints . . . 4

3 Meta-heuristics **6**
- 3.1 GRASP implementation . . . 6
  - 3.1.1 Constructive phase . . . 6
  - 3.1.2 Greedy cost function . . . 7
  - 3.1.3 Other problem specific details . . . 7
  - 3.1.4 Local Search . . . 7
- 3.2 BRKGA implementation . . . 9
  - 3.2.1 Chromosome structure . . . 9
  - 3.2.2 Decoder . . . 9

4 Testing, Tunning and Comparisons **10**
- 4.1 Benchmarking instances . . . 10
  - 4.1.1 The instance generation process . . . 10
  - 4.1.2 The medium and large instance sets . . . 11
- 4.2 Comparing Integer Linear Programing to Meta-heuristics . . . 12
- 4.3 Comparison: GRASP vs BRKGA . . . 12
  - 4.3.1 Tuning GRASP parameters . . . 13
  - 4.3.2 Tuning BRKGA parameters . . . 14
  - 4.3.3 Comparative results of meta-heuristics performance . . . 15

5 Conclusions **17**

A Appendices **18**
- A.1 Running the launcher . . . 18
- A.2 Parameter validation (LargeSet) . . . 18
  - A.2.1 Other options . . . 18
- A.3 Reporting . . . 18

Abstract

Optimization problems can appear in almost all situations on life, but there are special important cases where those problems appears, specially on the industry, since if we can achieve an optimal solution, we will improve the efficiency of the industrial process and then get lower production cost with the same (or better) efficiency. This improvement of the costs have an effect on the competitiveness of the companies and on the final quality of their products, including that this is an important money-saving factor.

The challenging part is when this kind of problems become really big, since then they have a attached really high computational complexity and cost, therefore we need some methods to face them.

To do so, we have two approaches, the always-optimal methods that obtain the optimal solution but taking into account every possible combination of the problems' solutions, as for instance Integer Linear Programming does. By the other hand we have those methods that concerns about fair execution times and are looking for a trade-off between acceptable execution times and the quality of the solution.

In this work we will propose two approaches in order to solve the scheduling of nurses in a hospital taking into account several constraints.

1 Introduction

Optimization problems can appear in almost all situations on life, but there are special important cases where those problems appears, specially on the industry, since if we can achieve an optimal solution, we will improve the efficiency of the industrial process and then get lower production cost with the same (or better) efficiency. This improvement of the costs have an effect on the competitiveness of the companies and on the final quality of their products, including that this is an important money-saving factor.

The challenging part is when this kind of problems become really big, since then they have a attached really high computational complexity and cost, therefore we need some methods to face them.

To do so, we have two approaches, the always-optimal methods that obtain the optimal solution but taking into account every possible combination of the problems' solutions, as for instance Integer Linear Programming does. By the other hand we have those methods that concerns about fair execution times and are looking for a trade-off between acceptable execution times and the quality of the solution.

In this work we propose two approaches in order to solve the scheduling of nurses in a hospital taking into account several constraints. The first one is using Integer Linear Programming and the second one by using metaheuristics. In the metaheuristics part we are focusing specifically in the Greedy Randomized Adaptative Procedure (GRASP) and the Biased-Random Key Genetic Algorithm (BRKGA).

Firstly we compare the efficiency in terms of solving time and wellness of ILP and metaheuristics over medium-sized problems.

Lastly, since large problems are intractable for ILP due to combinatorial explosion reasons, we compare the metaheuristics among them to compare such kind of problems.

1.1 Problem Statement

A public hospital needs to design the working schedule of their nurses. As a first approximation, we are asked to help in designing the schedule of a single day. We know, for each hour h, that at least demandh nurses should be working at the hospital. We have available a set of nNurses nurses and we need to determine at which hours each nurse should be working. However, there are some limitations that should be taken into account:

- Each nurse should work at least minHours hours.
- Each nurse should work at most maxHours hours.
- Each nurse should work at most maxConsec consecutive hours.
- No nurse can stay at the hospital for more than maxPresence hours (e.g. if maxPresence is 7, it is OK that a nurse works at 2am and also at 8am, but it not possible that he/she works at 2am and also at 9am).
- No nurse can rest for more than one consecutive hour (e.g. working at 8am, resting at 9am and 10am, and working again at 11am is not allowed, since there are two consecutive resting hours).

The goal of this project is to determine at which hours each nurse should be working in order to minimize the number of nurses required and satisfy all the aforementioned constraints.

1.2 Document Structure

The structure of this document is the following: The problem definition is done in chapter 2. Here it is explained the problem that is faced, which constraints is needed to take into account and what we want to optimize. At chapter 3 is explained the ILP model that has been developed, i.e. the decision variables and also the constraints with a mathematical nomenclature. After this chapter, at chapter 4 is explained how it has been used the heuristics approach in order to face with the problem, two meta-heuristics has been used: GRASP3 and BRKGA4 . After perform several executions for those approaches, a comparison in terms of time and quality of the result is done at chapter 5. Finally conclusions of the project are explained at chapter 6.

2 Integer Linear Programming

Integer Linear Programming is the first of the two methods that has been used in this project (see Introduction for more details about the problem). This method always finds the optimal solution without taking into account the amount of computational resources needed. This model is developed in CPLEX and the model implemented is described in the next sections.

2.1 Decision variables

- $w_{n,h}(\mathbb{B})$: This boolean variable specifies whether the nurse n works at the hour h (1) or not (0)
- $z_n(\mathbb{B})$: This boolean variable specifies whether the nurse n works during the shift or not.
  - ⋆ $zn = 1 \Rightarrow$ The nurse n works at least 1 hour, $\exists h, w_{n,h} = 1$
  - ⋆ $zn = 0 \Rightarrow \forall h, w_{n,h} = 0$
- $s_n(\mathbb{N})$: Positive integer variable specifying the hour in which the nurse n starts working, such that $w_{n,s_n} = 1$ and $w_{n,s_n-i} = 0$, $\forall i : 1 \leq s_n - i < s_n$
- $e_n(\mathbb{N})$: Positive integer variable specifying the hour in which the nurse n stops working, such that $w_{n,e_n} = 1$ and $w_{n,e_n+i} = 0, \forall i : e_n < e_n + i \leq 24$

2.2 Instance parameters

- $demand_h$: Array of integers, specifying the required number of nurses at hour h
- $nNurses$: Integer that specifies the number of available nurses to assign.
- $minHours$: Integer that specifies the minimum number of hours that a nurse must work if she works.
- $maxHours$: Integer that specifies the maximum number of hours that a nurse must work if she works.
- $maxConsec$: Integer that specifies the maximum number of consecutive hours that a nurse can work.
- $maxPresence$: Integer that specifies the maximum number of hours that a nurse can stay at the hospital.

2.3 Objective function

$$\text{Minimize} \sum_{n=1}^{nNurses} z_n$$

This objective function aims to minimize the number of working nurses, this means, minimize the number of z_n variables that are activated (with a value of 1), which is the main goal for our problem.

2.4 Constraints

- Set the z_n values correctly:

$$\forall n : 1 \leq n \leq nNurses, 24 \cdot z_n \geq \sum_{1 \leq h \leq 24} w_{n,h} z_n \leq \sum_{1 \leq h \leq 24} w_{n,h} \tag{1}$$

- At any hour h, at least demandh nurses must be working:

$$\forall h : 1 \leq h \leq 24, \sum_{1 \leq n \leq nNurses} w_{n,h} \geq demand_h \tag{2}$$

- Each nurse that works, must work at least minHours:

$$\forall n : 1 \leq n \leq nNurses \sum_{1 \leq h \leq 24} w_{n,h} \geq minHours \cdot z_n \tag{3}$$

- Each nurse that works, must work at most maxHours:

$$\forall n : 1 \leq n \leq nNurses \sum_{1 \leq h \leq 24} w_{n,h} \leq maxHours \cdot z_n \tag{4}$$

- Each nurse works at most maxConsec consecutive hours:

$$\forall n : 1 \leq n \leq nNurses, \forall h_1 : 1 \leq h_1 \leq 24 - maxConsec, \sum_{h_1 \leq h \leq h_1 + maxConsec} w_{n,h} \leq maxConsec \tag{5}$$

- Each nurse can stay in the hospital at most maxPresence hours:

$$\begin{aligned}
&\forall n : 1 \leq n \leq nNurses, e_n \leq 24 \cdot z_n, \\
&\forall n : 1 \leq n \leq nNurses, \forall h : 1 \leq h \leq 24, e_n \geq h \cdot w_{n,h}, \\
&\forall n : 1 \leq n \leq nNurses, s_n \geq 0, \\
&\forall n : 1 \leq n \leq nNurses, \forall h : 1 \leq h \leq 24, s_n \leq (h - 24) \cdot w_{n,h} + 24 \cdot z_n, \\
&\forall n : 1 \leq n \leq nNurses : \\
&e_n - s_n + 1 - (2 \cdot 24) \cdot (1 - z_n) \leq maxPresence \cdot z_n
\end{aligned} \tag{6}$$

- Each nurse can rest at most one consecutive hour:

$$\begin{gathered}
\forall n : 1 \leq n \leq nNurses, \forall h : 2 \leq h \leq 22, \forall M : M \geq 24 : \\
M - M \cdot w_{n,h-1} + M \cdot w_{n,h} + M \cdot w_{n,h+1} \geq \sum_{h+1 \leq h_i \leq 24} w_{n,h_i}
\end{gathered} \tag{7}$$

3 Meta-heuristics

3.1 GRASP implementation

3.1.1 Constructive phase

The GRASP meta-heuristic has a constructive phase that is concerned to build up a feasible solution. This phase can be deterministic or include a certain amount of randomness by controlling a paremeter value α. This means that for every execution, different solution could emerge. We use the basic GRASP construction phase described in [1]. The specific parts of the implementation are the initialization of the candidate set C, implemented in the function *initializeCandidates* depicted in *Algorithm* 1, and the greedy cost function shown in the next subsection.

Algorithm 1: initializeCandidates

```
Input: nNurses, hours, schedule_constraints
Output: candidate list C
1  $C \leftarrow \emptyset$
2  foreach $h \in hours$ do
3      $E \leftarrow initializeEmptySchedules(10, hours, schedule_constraints)$
4      foreach $hindex \in [h+1, hours]$ do
5          $modulo\_param = 2$
6          foreach $e \in E$ do
7              if $hindex - h \bmod modulo\_param > 0$ then
8                  addWorkingHour(e, hindex)
9                  if notValidConstraints(e) then
10                     removeWorkingHour(e,hindex)
11                     addRestingHour(e, hindex)
12                 end
13             end
14             else
15                 addRestingHour(e, hindex)
16                 if notValidConstraints(e) then
17                     removeRestigHour(e,hindex)
18                     addWorkingHour(e, hindex)
19                 end
20             end
21             if notValidConstraints(e, hindex - h) then
22                 $E \leftarrow E \cap e$
23             end
24             $modulo\_param += 1$
25         end
26     end
27     $C \leftarrow C \cup E$
28 end
29 return <C>
```

The ComputeCandidateElements function takes as input the total number of nurses, the number of hours to schedule and the rest of the constraints(maximum presence hours, consecutive hours, total hours and minimum hours). The output is a list of multiple schedules that each nurse can be assigned to. A schedule is the list of hours in which a nurse works must work.
First, the algorithm initializes an empty candidate set. Then, for each hour in the schedule, it creates 10 different types of schedules beginning a this specific hour(line3). The difference between the 10 types of schedules is the compactness of the working hours. This is controlled by a parameter used to do the modulo with current hour index in the built schedule. This allows to create from the most compact schedule with all working hours consecutive until the constraints allows to do it ($hindex - h \bmod hours + 1$), to the most sparse schedule consisting of alternating working and resting hours (using $hindex - h \bmod 2$). The different schedules, started at different hours are built incrementally by adding work or rest hours depending on the modulo parameter (line 7), and always taking into consideration the validity of the resulting partial schedule (line 9, 16), in which

case the validity is temptatively fixed. If no more options remain and the schedule becomes invalid for the constraints of the problem(line 21), it is removed from the set E (line 22) and so it is not later saved to the candidate set C (line 27).

3.1.2 Greedy cost function

As all the nurses are equal in this scenario, there's only one thing that the greedy cost function can determine, the number of hours of demand that a single nurse schedule contributes to. We are able to do this because all schedule candidates produced are valid (they follow the constraints). We conside e to be a candidate schedule for a single nurse, being $e_h = 1$ if the nurse has to work or 0 otherwise. We also use a big constant number K, that should be bigger than the value *hours* (for example $nNurses$) and we consider $remaining\_demand_h$ to be the demand that is not covered by any other schedule that is present in the solution at the hour h .

$$gc(e) = K - \sum_{h=1}^{Hours} remaining\_demand_h \times e_h$$

3.1.3 Other problem specific details

There are some specific modifications of the GRASP constructie phase of [1] applied to this problem. After updating the candidate set, in [1] page 2, we test the feasibility of the updated solution in each iteration of the constructive phase, and in the case we are having a feasible solution, we leave the loop. If not, we continue looping until no candidate schedules are available or no more candidate schedules can be added (all nurses have a schedule). A solution is feasible if for each hour, the demand is less or equal to the number of nurses working at this hour. Another improvement introduced in the basic version of the algorithm, is the fact that instead of removing candidate elements(single nurse schedules) from the candidate set and adding them to the solution, we generate a basic list of possible candidates and each time a nurse is assigned one of them, we don't remove it from the candidate set. What we do is recompute each time the greedy cost of each candidate when the solution is updated. This is possible thanks to the fact that all nurses are equal under the problem assumptions and constraints. That way we reduce the number of candidates that we have to generate and sort.

3.1.4 Local Search

Once the constructive phase ends, the local search tries to improve the given solution by looking at its neighbourhood,i.e. the near solutions will be searched. We show the pseudocode in (*Algorithm* 2). We perform a lightweight local search after every GRASP constructive phase, and then when *maxIterations* of GRASP constructive phases have been performed, we execute a more intensive local search.

Algorithm 2: Local Search

```
Input: solution, nNurses, hours, demand
Output: solution
1  $sol \leftarrow removeExceedingWorkingHours(solution)$
2  improved = True
3  while $improved$ do
4      improved = False
5      $G \leftarrow findValidScheduleGaps(sol)$
6      while $G \neq \emptyset$ do
7          foreach $n \in nNurses$ do
8              $sol' \leftarrow sol$
9              $G_{aux} \leftarrow G$
10             foreach $h \in hours$ do
11                 if $sol'(n, h) \neq 1$ then
12                     continue
13                 end
14                 $sol' \leftarrow swapHourAssignment(sol', n, G(h), h)$
15                 $G \leftarrow removeFirstScheduleGapAtHourH(G, h)$
16             end
17             if $isValidSolution(sol')$ and $improvesCost(sol', sol)$ then
18                 $sol \leftarrow sol'$
19                 Improved = True
20             end
21             else
22                 $G \leftarrow G_{aux}$
23             end
24         end
25     end
26 end
27 solution ← sol
28 return solution
```

The local search algorithm starts by removing any work assignment that is not needed to fulfill the demand of the problem (line 1). This will make room for later swaping of hour assignments (and sometimes frees some nurses). Then we initialize the improvement while loop, a loop that will stop when no nurse can be freed anymore. Inside the loop(line 5), we first save all the gaps of the solution. That means all hours that a working nurse is resting but that he/she could work and its schedule would still be valid. The gap list G contains for each possible hour of the schedule, a list of nurses that have a valid gap at these specific hour. A new loop is created until there's no more valid gap left (line 6). Inside this loop, we try, for each nurse, to assign its working hours to the first nurse that is listed in the gap list for each hour(line 14, 15). If after swaping all the hours of a nurse the resulting solution is valid and improves the solution, then the solution is updated (line 17). If not, we undo all the changes to the gap list (line 22) and we don't update the current solution. At the end, the returned solution contains either the same solution as the beginning, or a new solution with less working nurses but all valid and fullfiling the demand at each hour.

To transform the firts improvement local search shown previously into an intensive local search, we can add room for performing iterations of the local search even if the solution has not improved. We could limit that with a variable called $failedIterations$. Thats the approach we choose to perform on the last localsearch we perform after the main GRASP loop (see first algorithm in [1] on page 2). Usually, there's also the possibility to implement a first improvement local search and a best improvement local search. There is no such possibility with this implementation. We would require to start swaping nurse hour assignments from different nurses instead of following always the same order. This turned to be too costly in terms of algorithmic cost and we decided to go with this simpler approach.

3.2 BRKGA implementation

3.2.1 Chromosome structure

Since in this problem all the nurses are equal in terms of schedule, to exploit the diversification of the chromosoem we have chosen between two approaches. The firsts is to assign each gene of the chromosome in order to each hour of the schedule (chromosome length *hours*). This ways we can sort the hours of the schedule and assign nurses to hours in that order. The other choice is instead to define an excess of working nurses for each hours, that is, to increase the demand of each hour randomly by the chromosome. We choose the second approach as it appears to diversify more in the initial tests that we have performed (around 20% of different fitness between individuals in each breed versus only around 5% for the first approach).

Algorithm 3: BRKGA Decoding algorithm

```
Input: chromosome, hours, demand, nNurses
Output: demand
1 i = 0
2 foreach gene ∈ chromosome do
3     if $gene < 0.2$ then
4         $new\_hourly\_demand = ceil(demand_i \cdot 0.8 \cdot nNurses)$
5         $demand_i = new\_hourly\_demand$
6     end
7     $i+ = 1$
8 end
9 return demand
```

3.2.2 Decoder

Algorithm 4: BRKGA Decoder algorithm

```
Input: population, nNurses, hours, demand, constraints
Output: population
1 foreach individual ∈ population do
2     new_demand ← $decoding(individual, hours, demand, nNurses, constraints)$
3     population ← $assignNurses(solution, nNurses, hours, new\_demand)$
4 end
5 return population
```

As illustrated in *Algorithm* 4, the decoder simply decodes each individual chromosome with *Algorithm* 3, and calls *assignNurses* to assign the nurses according to the newly computed *new_demand*. Each solution is stored in the population with its corresponding fitness already computed in the *assignNurses* function (*Algorithm*

5).

Algorithm 5: assignNurses

```
Input: solution, nNurses, hours, new_demand, constraints
Output: solution
1  foreach $h \in hours$ do
2      $mustWorkList, canWorkList = computeCandidateAssignments(solution, h, constraints)$
3      foreach $n \in mustWorkList$ do
4          $assignWorkingHour(solution, n, h)$
5          $demand_h \leftarrow updateRemainingDemand(solution, h)$
6      end
7      foreach $n \in canWorkList$ do
8          if $demand_h > 0$ then
9              $assignWorkingHour(solution, n, h)$
10             $demand_h \leftarrow updateRemainingDemand(solution, h)$
11         end
12     end
13 end
14 if $Feasible(solution)$ then
15     $solution(fitness) \leftarrow computeFitness(solution)$
16 end
17 else
18     $solution(fitness) \leftarrow \infty$
19 end
20 return solution
```

As shown in *Algorithm* 5, this algorithm assigns nurses to the schedule according to the demand and the constraints. For each possible hour in the schedule, in order, the algorithm selects the nurses that should work at this specific hour according to the constraints, and the nurses that could work at this specific hour according to the constraints (line 2). The function *computeCandidateAssignments* walks through all the nurses, assigning the current hour and verifying if the nurse schedule is valid according to the constraints. From that "canWorkList", it walks through all the nurses from that list, removing the assignment in the specific hour and verifying if the schedule would be valid or not. That way it creates the second "mustWorkList" with nurses that must work to have valid schedules, removing them from the first list in order to have two disjoint sets. Once the two lists are set, it first assigns all the nurses that must work to have a valid schedule (lines 3 to 5). Each time, the remaining demand is updated (line 5). Then it walks throught the list of nurses that can work and assign the hour if and only if the remaining demand in that specific hour is positive (line 8). The final solution is then updated with its fitness, which is the same as the cost or number of nurses that work any hour during the schedule. If the solution is not feasible, that means that the demand is not fulfilled in each hour, then the fitness is set to be infinite or big enough.

4 Testing, Tunning and Comparisons

4.1 Benchmarking instances

This section explains the two sets of benchmarks used.The medium size instance set consists of 38 small or medium size instances, and the large size instance set consists of 200 problem instances. The medium size set is used to compare the ILP model versus the Metaheuristics models. The large size set is used to test parameter values in order to find the best performing setup for each BRKGA and GRASP model, and finally to compare them solving one large problem instance.

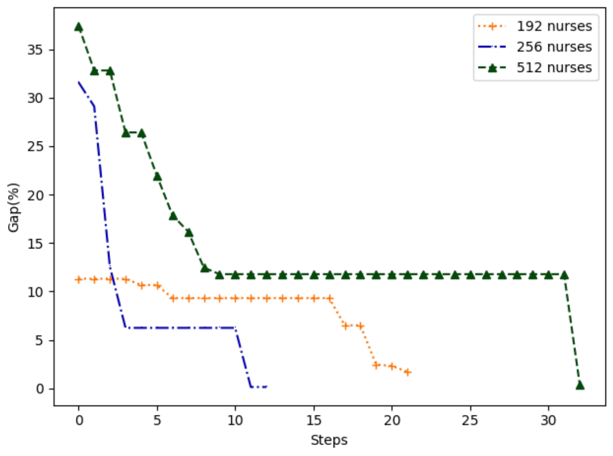
4.1.1 The instance generation process

The algorithm used to generate problem instances is based on generating random but valid schedules of nurses and then adding the number of nurses working at each hour to compute the demand. This approach assures that the problem instance will always have a solution. Besides that, the algorithm places the schedules close to few selected hours of the schedule (adding some variability). The input values of the algorithm are all the variables of the problem instance, the selected hours of the schedule to place nurse schedules around and some proportion of extra nurses to add to the problem without using them to create the demand.

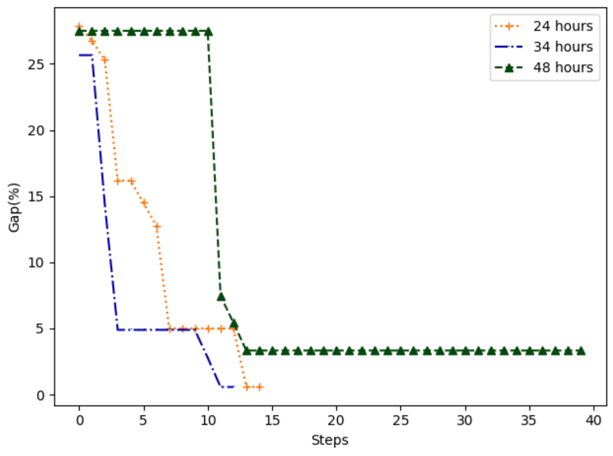
4.1.2 The medium and large instance sets

To decide how the size of a problem is determined, we choose to reference the time it takes for the ILP model to solve the problem instance using the Cplex solver. We generate and solve a series of instances with the ILP until we have around 20 instances that take 60 minutes or less to solve by ILP. Then, as we cannot perform the same procedure to generate the large instance set, we execute a series of experiments to determine the influence of the problem variables to the steps the ILP model has to do to decrease the gap between the best integer and the best bound.

The results are shown in figure 1, figure 2 and figure 3 page 11. The table 1 page 12 summarizes the findings and shows how we applied them to create the large benchmark instance set.

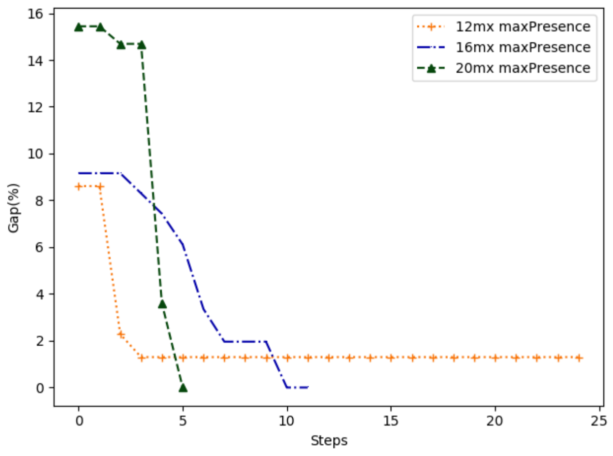


(a) Evolution of Gap of the ILP model with different number of nurses

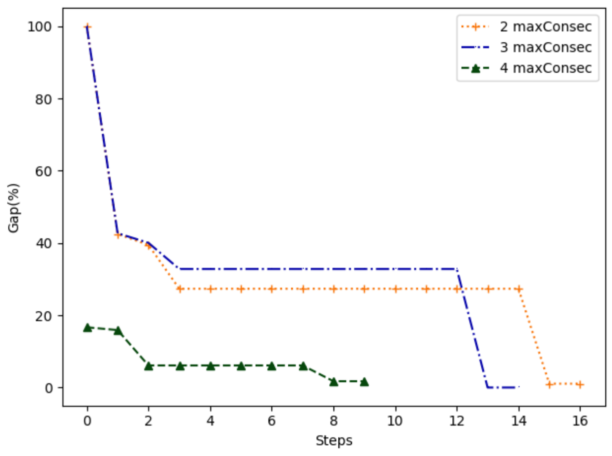


(b) Evolution of Gap of the ILP model with different number of hours

Figure 1: Evolution of Gap of the ILP model with different values of number of nurses (a) and number of hours (b).

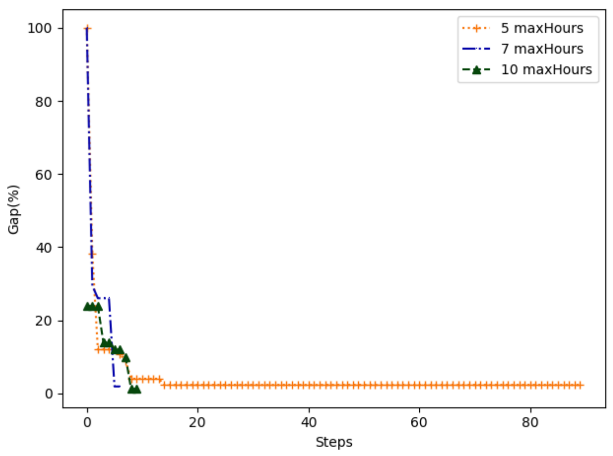


(a) Evolution of Gap of the ILP model with different values of maxPresence parameter

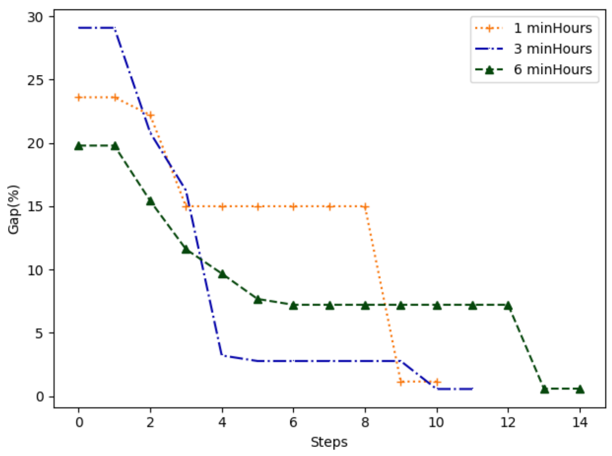


(b) Evolution of Gap of the ILP model with different values of maxConsec parameter

Figure 2: Evolution of Gap of the ILP model with different values of maxPresence (a) and maxConsec (d).



(a) Evolution of Gap of the ILP model with different values of maxHours parameter



(b) Evolution of Gap of the ILP model with different values of minHours parameter

Figure 3: Evolution of Gap of the ILP model with different values of maxHours (a) and minHours (d).

Problem Variable	**Effect on problem size when increasing**	**Medium size set** 38 instances	**Large size set** 199 instances
$nNurses$	increases	64	64 to 4096
$hour$	increases	24	24, 48, 72
$maxPrensce$	decreases	16	8 to 27
$maxConsec$	decreases	5	4 to 13
$maxHours$	decreases	4 to 10	2 to 12
$minHours$	increases	1 to 5	1 to 9

Table 1: Benchmark datasets and variables influence on problem size

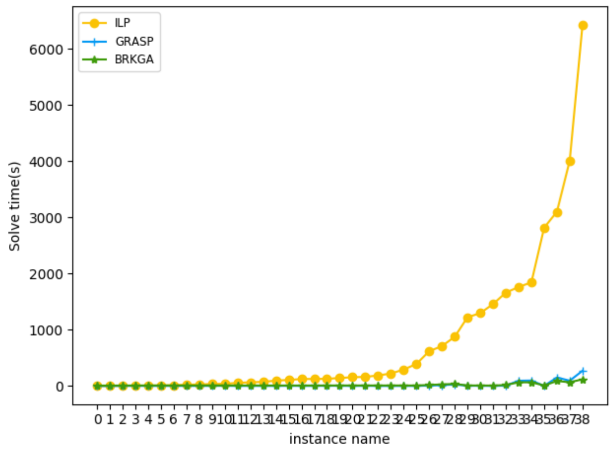
4.2 Comparing Integer Linear Programing to Meta-heuristics

In this section, we are going to compare the execution results of the Cplex ILP solver and the two meta-heuristics algorithms. We are going to solve the problem instances of the medium size data set with our three models and plot the results in terms of solving time and objective function. All executions have been performed on the same computer. The parameters for the BRGKA and GRASP algorithms are respectively kept with the same values during all executions. For this experiment, the size of the problems will be determined by the time it takes to the ILP model to find the optimal solution.

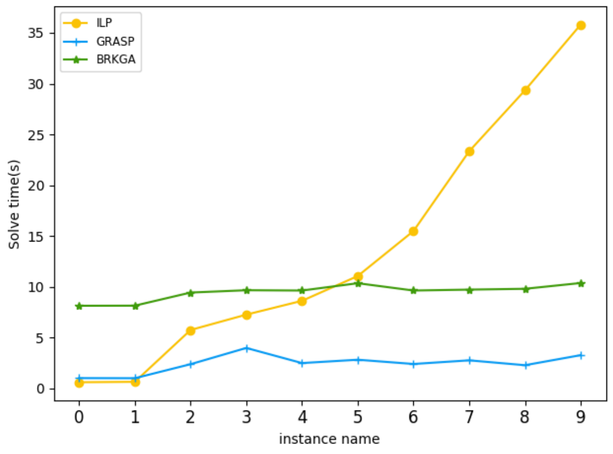
The figure 4 on page 13, shows that the ILP execution times increase exponentially as the size of the problem instance increases, whereas the GRASP and BRKGA solving times increase much more slowly. We observe that for the bigger instances, the GRASP algorithm produces worse results (bigger objective function values), whereas the BRKGA produces results very close to the optimum objective function found by the ILP solver.

4.3 Comparison: GRASP vs BRKGA

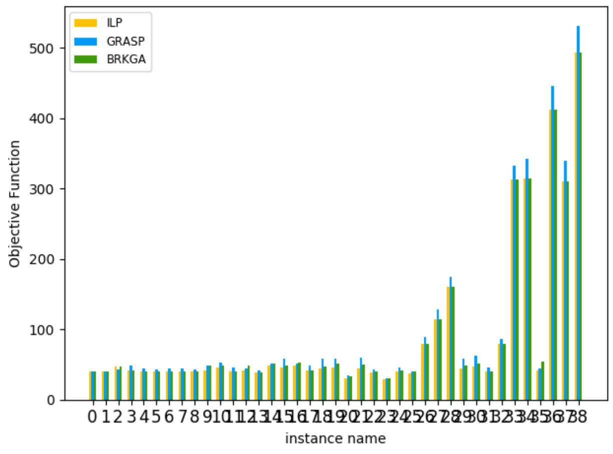
In this section we are going to compare the performance of both metaheuristic algorithms applied to the assignment problem.
In order to do that, we are first going to choose the best parameters for each algorithm using the large set of problem instances described previously to execute several times each algorithm. The procedure consists of executing the same subset of samples drawn from the large set of problem instances for different proposed values of each parameter tested. The executions can be performed in different computers since the time is not important, only the objective function. The only caveat, is that all parameter values have to solve the same set of instances, since we use the average of objective functions. Otherwise the results would not be coherent from one parameter value to another. We record the objetive function for each execution and compute its average for each parameter value. Then we plot its evolution. We choose the parameter that produces the minimum



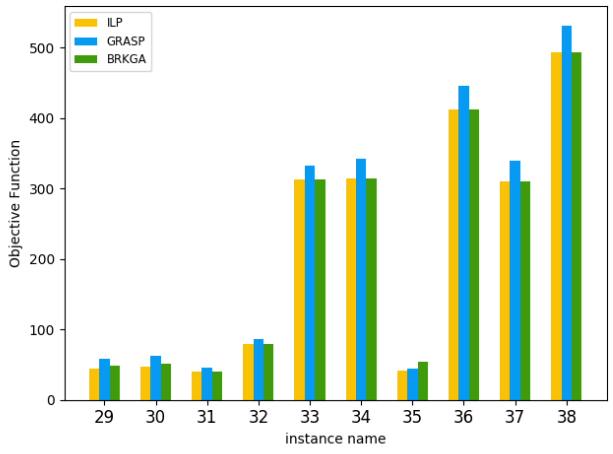
(a) Solving times for 38 instances of the mediume set



(b) Solving times for the first 10 instances



(c) Objective function values for 38 instances of the medium set



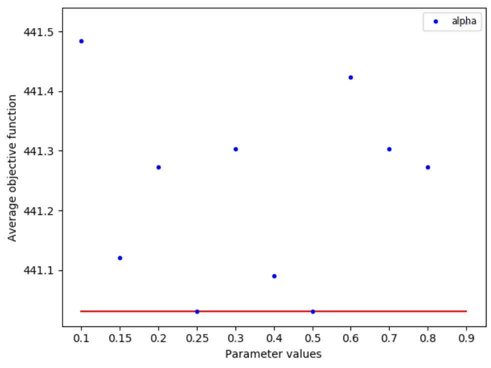
(d) Objective function values for the last 10 instances

Figure 4: Solving times in seconds (a, b) and objective function values (a, d) for the instances of the medium size set solved by ILP, GRASP and BRKGA.

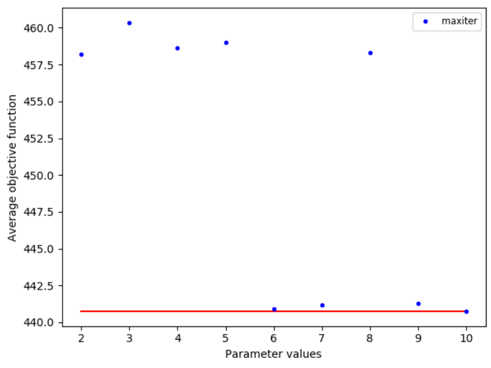
objective function average.
Finally, we will use the best parameter values for each algorithm, to compare the performance of both algorithm solving the same problem instance.

4.3.1 Tuning GRASP parameters

For the parameters of the GRASP algorithm, we have tested the α, the $maxIterations$ of the main loop and the $failedIterations$ for the final and more intensive local search. The results of the tests are shown in figure 5 on page 14 and figure 6 on page 14.



(a) GRASP Parameter α

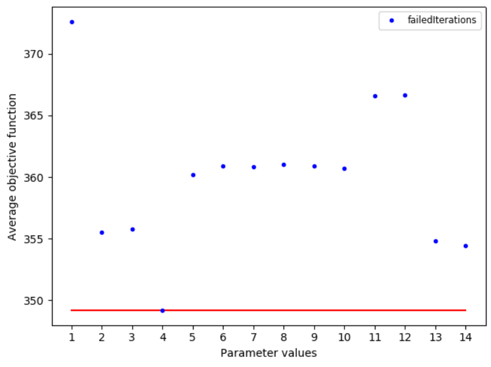


(b) GRASP Parameter $maxIterations$

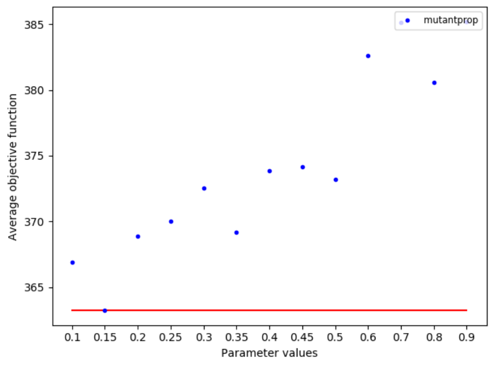
Figure 5: Average objective function for different (a) α and (b) $maxIterations$ values of the GRASP model

4.3.2 Tuning BRKGA parameters

For the parameters of the BRKGA algorithm, we have tested the number of generatons (*generations*), the number of individuals in the population (*population*), the inheritance probability (*inheritance*), the proportion of elite individuals in each generation (*eliteprop*) and the proportion of mutant individuals in each generation (*mutantprop*). The results of the tests shown in figure 6 on page 14, figure 8 on page 15 and figure 7 on page 15 .

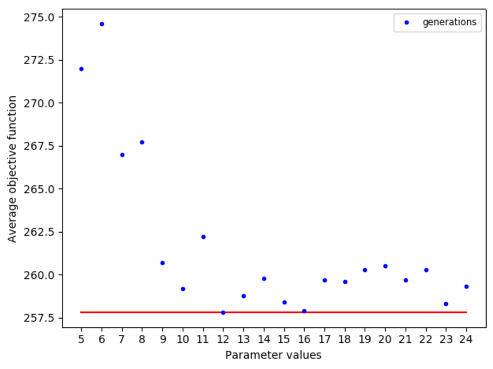


(a) GRASP Parameter $failedIterations$

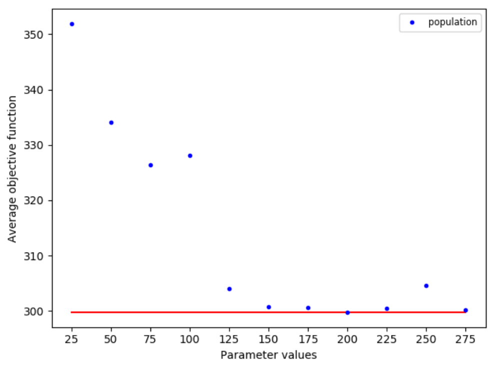


(b) BRKGA Parameter $mutantprop$

Figure 6: Average objective function for different (a) $failedIterations$ of the GRASP model and (b) $mutantprop$ values of the BRKGA model

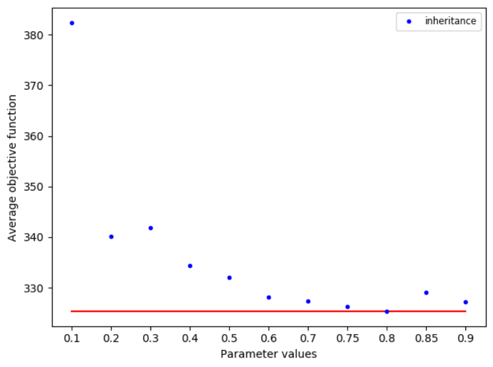


(a) Parameter *generations*

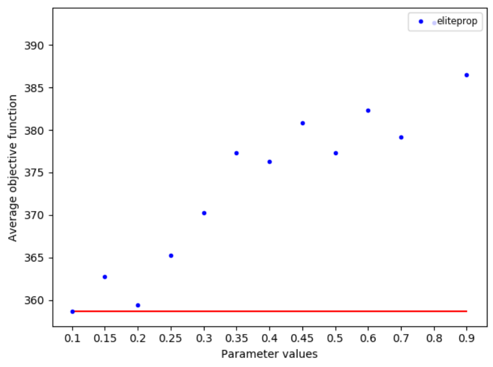


(b) Parameter *population*

Figure 7: Average objective function for different (a) *generations* and (b) *population* values for the BRKGA model



(a) Parameter *inheritance*



(b) Parameter *eliteprop*

Figure 8: Average objective function for different (a)*inheritance* and (b) *eliteprop* values for the BRKGA model

4.3.3 Comparative results of meta-heuristics performance

Finally, we execute the same problem instance from the large set with both GRASP and BRKGA models. The selected problem instance is the following: **i-ng-60-4096-2560-24h-16mxP-4mxc-10mxH-1mnH-3Cnt-20180109_08-04-50394.dat**

In the figure 9 we observe that it takes the GRASP almost two more hours than BRKGA to finish. However, it finds a better local optimum (3190) than BRKGA(3209), except that both models find a local optimum around 3200 and the optimum is 2560. The evolution of the objective function for the GRASP model shows a series of steps, that are combinations of the result of the Randomized Greedy constructive algorithm and then the Localsearch (which in most cases does not improve the solution). Then, around 2 hours from the beginning the objective function of the GRASP model starts to decrease. This is where the final intensive Localsearch algorithm is improving the solution (from the best pure Greedy or randomized Greedy constructive result). Because of the construction of the graph, the last Localsearch seems to improve from the last iteration of the Grasp (Constructive plus Local search) but in fact it starts from the pure Greedy result at the very beginning. The final Localsearch does 36 iterations, starting at the best incumbent 3225 and decreasing slowly from 3222 to finally get an objective function value of 3190.

As we can observe in the chart, the BRKGA model finds the a local optimum in the fourth generation, but as it is setup to do twenty generations, it continues for 30 more minutes. This is caused by the fact that we have tested the parameters in isolation. We have not taken into account their interactions and strange results can appear like this early optimal finding of the BRKGA. An improvement that could be implemented is, as is done in the intensive localsearch, to introduce a limited number of non improving iterations upon which the algorithm stops. Looking at the diversity for each breed of individuals of the BRKGA model, we find that it starts with

GRASP

- $\alpha = 0.25$
- $maxIterations = 10$
- $failedIterations = 4$

BRKGA

- $generations = 12$
- $population = 200$
- $inheritance = 0.8$
- $eliteprop = 0.1$
- $mutantprop = 0.15$

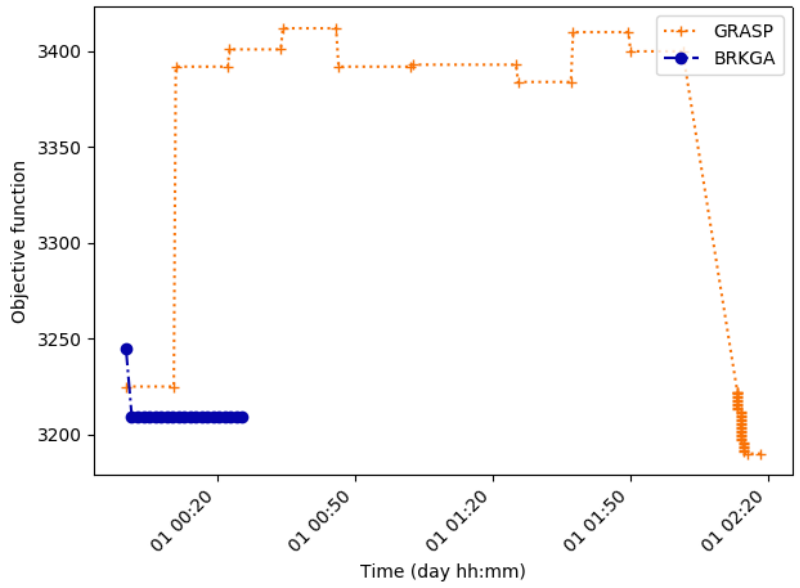


Figure 9: Objective function evolution by GRASP and BRKGA models using the best parameter setup.

a rate of 30% (61 different fitness values out of 200 individuals), increases to 41% in the third generation then decreases to around 10% and stays there. In the last generations it reaches 17%, but the objective function stays the same as in the third generation where it finds the local optimum 3209.

In terms of intensification we can say that the final Localsearch of the GRASP algorithm does a good job, as well as the mating in the BRKGA's decoder. However the lightweight Localsearch during the GRASP iterations hasn't found improvements. In terms of diversification, this problem instance hasn't yield very impressive results. For example, the GRASP starts from 9 different constructive phases but they are close in terms of objective function. Moreover, the randomized constructive phases do not improve the pure Greedy approach so the space exploration capabilities of GRASP are not exploited as they could be.

5 Conclusions

In this project we developed an ILP model and two meta-heuriscs in order to find out the optimal (for ILP) and good (for meta-heuristics) nurse schedules in a hospital taking into account several constraints.

On the first hand, the Integer Linear Programming part has been really challenging to end up with a really simplified model that takes into account every constraint. We also saw how the cost of solving a problem increases exponentially with the problem size, whereas with the meta-heuristic models the cost increases much more slowly allowing use to solve much more bigger instances.

On the second hand, the meta-heuristics has been also a challeng to build up with a reasonable greedy function, that is in fact the core of all applied meta-heuristics here. But the more challenging part has been to get a good set of inputs for the experiments, specially the largest of them. It has been so difficult since the complexity is not only a matter of number of nurses but also another factors that determine the problem context, but finally we end up with good inputs by tunning the instance generator. It is also difficult to set the parameters of the meta-heuristic algorithms to exploit the diversification and intensification options that they poses.

Finally, we demonstrated how simpler algorithms can be combined together to attack difficult problems and obtain a satisfactorial solution.

References

[1] M.G.C. Resende and C.C. Ribeiro. *Greedy randomized adaptive search procedures.* In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, pages 219–250. Kluwer Academic Publishers, 2003

[2] JF Gonçalves, MGC Resende *Biased random-key genetic algorithms for combinatorial optimization* In - Springer, Journal of Heuristics, 2011

A Appendices

A.1 Running the launcher

To solve instances of the problem with the models run, from ./Tools/Launcher:

For ILP (calls oplrun from commandline) python launcher.py –solver ILP

For GRASP python launcher.py –solver grasp For BRKGA

python launcher.py –solver brkga

These commands solve all instances from the default folder ./Instances/Pending and save the results in a json format to ./Results/Pending Options:

Instances folder and results folder (paths must be relative to ./Tools/Launcher) : python launcher.py –solver grasp –instances ../../Instances/Final/MediumSet –results ../../Results/Final/MediumSet

GRASP parameters: python launcher.py –solver grasp –iterations 5 –alpha 0.15 –ls best

BRKGA parameters: python launcher.py –solver brkga –generations 1 –eliteprop 0.3 –mutantprop 0.2 –population 2 –inheritance 0.1 –decoder <hini,horder,hexcess>

A.2 Parameter validation (LargeSet)

$cdTools/Launcherpythonmegalauncher_grasp.py../../Instances/Final/LargeSet_2 0180106/$

Quick demo of the files and dirs created in ../../Results/Final/LargeSet_20180103/. It does not execute any solver, just time.sleep(0.5) cd Tools/Launcher python megalauncher.py –demo ../../Instances/Final/LargeSet

Real execution cd Tools/Launcher python megalauncher.py ../../Instances/Final/LargeSet See results in ../../Results/Final/LargeSet_20180103/ See progress files in ../../Results/Progress/

A.2.1 Other options

Other options that can be used are to use the ./Metaheuristics/main.py file from it's 'main' section, calling any of the "entry points" of the Metaheuristics algorithms.

A.3 Reporting

mediumSetGraph.py, takes all json files in a folder and creates 2 graphs, one comparing the solving times for differents instances for the 3 algorithms (ILP, GRASP and BRKGA). It expects to find json files with results from executions of the 3 algorithms in the indicated folder: python mediumSetGraph.py ../../Results/MediumSet

summary_of_executions.py, example file that reads json results files from a folder and plots all the resutls in a graph

classifier.py script used to organize the executions of instances by the ILP, it saves the instances files (.dat) to the Instances/Final/ with a name that reflects the time it takes for the ILP to solve it.