AMMM Course Project Report

Nurses in Hospital

Asaf Badouh, Alexander Parunov

January 7, 2018

1 Problem Statement

A public hospital needs to design the working schedule of their nurses. We know for, for each hour h, that at least $demand_h$ nurses should be working. We have available a set of *nNurses* and we need to determine at which hours each nurse should be working. However following constraints must be met:

1. Each nurse should work at least *minHours* hours.
2. Each nurse should work at most *maxHours* hours.
3. Each nurse should work at most *maxConsec* consecutive hours.
4. No nurse can stay at the hospital for more than *maxPresence* hours (e.g. is *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).
5. 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 is to determine at which hours each nurse should be working in order to **minimize** the **number of nurses** required and satify all above given constrains.

2 Integer Linear Model

Before defining the model and formal solution, set of parameters and decision variables that are used in model must be defined.

2.1 Parameters

- *nNurses:* Int - number of nurses
- *nHours:* Int - number of hours
- *minHours:* Int - minimum hours each nurse should work
- *maxHours:* Int - maximum hours each nurse can work
- *maxConsec:* Int - maximum consecutive hours each nurse can work
- *maxPresence:* Int - maximum number of hours each nurse can be present
- $demand_h[nNurses]$: Int - demanded number of working nurses at each hour, indexed h

2.2 Decision Variables

- $works_{n,h}[nNurses][nHours]$: Boolean - Nurse n works at hour h
- $WA_{n,h}[nNurses][nHours]$: Boolean - Nurse n works after hour h
- $WB_{n,h}[nNurses][nHours]$: Boolean - Nurse n worked before hour h
- $Rest_{n,h}[nNurses][nHours]$: Boolean - Nurse n rests at hour h. However, nurse should have worked before and should work after the resting hour h. Otherwise it's not considered as resting hour.
- $used_n[nNurses]$: Boolean - Nurse n is working

2.3 Objective Function

After defining all parameters and decision variable we might proceed with definition of objective function and formal constaints. In addition to that,for clarity, we denote set of *nHours* as H, and set of *nNurses* as N. So, since the goal of project is to have as few working nurses as possible satisfying $demand_n[nNurses]$ and all constraints, the objective function is:

$$\min \sum_{n=1}^{N} used_n \tag{1}$$

2.4 Constraints

Solution of problem must respect all constraints given in Section 1. Moreover, ILOG model direcly resembles formally stated constraints. Which means, after formally defining constraints, we may obtain an Integer Linear solution. So constraints are following:

Constraint 1: On each hour h at least $demand_h$ nurses should work.

$$\sum_{n=1}^{N} works_{n,h} \geq demand_h, \forall h \in H \tag{2}$$

Constraint 2: Each nurse n should work at least $minHours$ minimum number of hours.

$$\sum_{h=1}^{H} works_{n,h} \geq used_n \times minHours, \forall n \in N \tag{3}$$

Constraint 3: Each nurse n can work at most $maxHours$ maximum number of hours.

$$\sum_{h=1}^{H} works_{n,h} \leq used_n \times minHours, \forall n \in N \tag{4}$$

Constraint 4: Each nurse n can work at most $maxConsec$ maximum consecutive hours.

$$\sum_{j=i}^{i+maxConsec} works_{n,j} \leq used_n \times maxConsec, \forall n \in N, \forall i \in [1, nHours - maxCosec] \tag{5}$$

Constraint 5: No nurse n can stay at the hospital for more than $maxPresence$ maximum present hours. In other words, if the nurse worked at hour h, he/she cannot work after $h + maxPresence$ hour.

$$WB_{n,h} + WA_{n,h+maxPresence} \leq 1, \forall n \in N, \forall h \in \{h \in H | h \leq nHours - maxPresence\} \tag{6}$$

Constraint 6: No nurse n can rest for more than one consecutive hour.

$\forall n \in N, \forall h \in \{h \in H | h \leq nHours - 1\}$

$$WA_{n,h} \geq WA_{n,h+1} \tag{7}$$

$$WB_{n,h} \leq WB_{n,h+1} \tag{8}$$

$$Rest_{n,h} + Rest_{n,h+1} \leq 1 \tag{9}$$

In order to connect $WB_{n,h}, WA_{n,h}$ and $Rest_{n,h}$ decision variable matrices with a solution matrix $works_{n,h}$, we need to construct following logical equivalence:

$$Rest_{n,h} == (1 - works_{n,h}) - (1 - WA_{n,h}) - (1 - WB_{n,h}) \tag{10}$$

Which means:

If $Rest_{n,h} = 1$, then $(1 - works_{n,h}) - (1 - WA_{n,h}) - (1 - WB_{n,h}) = (1 - 0) - (1 - 1) - (1 - 1) = 1$.

If $Rest_{n,h} = 0$, then $(1 - works_{n,h}) - (1 - WA_{n,h}) - (1 - WB_{n,h}) = (1 - 0) - (1 - 1) - (1 - 0) = 0$

If feasible solution exists, then we get filled matrix $works_{n,h}$ and array of used nurses $used_n$ with minimized value of objective function $\min \sum_{n=1}^{N} used_n$. Which means this optimization problem is solved.

3 Metaheuristics

Integer Linear program guarantees to find an optimal solution if any feasible solution exists. However, this can be quite time consuming for medium and big size problems. That's why it's necessary to introduce some heuristics and solve this combinatorial optimization problem using them. Which assum-

ingly should give a worse solution than OPL, but faster. We will consider two meta-heuristics: **GRASP** and **BRKGA**.

3.1 GRASP

The essence of GRASP is to select a candidate for solution element from a set of candidate elements using some criteria, which will be discussed later. First of all we need to generate that candidate set. The working nurse might be viewed as a boolean array, where *1* means nurse works on that hour, and *0* means nurse doesn't work. An example nurse boolean array might look like that: [0, 1, 1, 1, 1, 0, 0, 0]. And this working nurse is exactly the candidate element for solution. In order to increase time efficiency and solution accuracy, we generate all possible elements for hours $h \in [1, 19]$ and save it locally. Then before starting solving problem, we filter that set of candidate elements so that it corresponds to specific constraints, given by parameters. Even if the set doesn't exist it still takes around 8 secs to generate all $2^{19} - 1 = 524287$ possible elements for hour $h = 19$. And after initial generation, we can load elements from computer. This will improve speed of program. However, in case $h \geq 20$, generating all possible candidate elements and storing them locally is highly time and space consuming. So instead, we temporarily generate randomly $4 \times nNurses$ feasible candidate elements, respecting all given constraints. And then use this set to solve our problem. In case of $nNurses = 1800$, it takes around 8 secs to generate this set of feasible element solutions. Our **GRASP** just like any other is divided into two phases *Constructive Phase* and *Local Search Phase*, with modifications of phases themselves.

3.1.1 Constructive Phase

1: **function** CONSTRUCT(gc(.),α)
2: Initialize candidate set C;
3: $\underline{s} = min\{g(t)|t \in C\}$
4: $\overline{s} = max\{g(t)|t \in C\}$
5: $RCL = \{s \in C|gc(s) \leq \underline{s} + \alpha(\overline{s} - \underline{s})\}$
6: Select candidate element x, at random, from RCL
7: **return** x
8: **end function**

So instead of classical **GRASP** *Constructive Phase*, where candiate solution is constructed, we return one candidate element for overall solution. In most of the cases random candidate element is not the best, so we perform a *Local Search* on returned candidate element x.

3.1.2 Local Search Phase

1: **function** LOCALSEARCH(x, C, α, $nHours$)
2: $maxDistance = \sqrt{nHours} \times (1 - \alpha)$
3: $N(x) = \{t \in C | dist(x, t) \leq maxDistance\}$
4: **for all** $n \in N$ **do**
5: **if** $gc(n) < gc(x)$ **then**
6: x = n
7: **end if**
8: **end for**
9: **return** x
10: **end function**

LOCALSEARCH function will return the best element in a neighborhood of candidate element, where neighborhood is defined by Euclidean distance from that element. The radius of neighborhood is also determined by α parameter. In case $\alpha = 0$, we get the greediest approach, which is the best and slowest at the same time.

The difference between classical **GRASP** and **ours**, is that we don't construct the set of candidate solutions, and then perform *Local Search* on that set. But rather randomly return 1 element from constructed RCL and apply local search on 1 element. This will reduce execution time, while returning same locally optimal candidate element. If we followed classical **GRASP**, we would construct full solution, then apply *Local Search* to modify some of the nurses in search neighborhood. This makes solution exponentially complex. While our method reduced the solution complexity to quadratic in the worst case, while modification of nurses still remains locally optimal. To conclude **GRASP** let's introduce our *Greedy Cost function*

3.1.3 Greedy Cost Function

1: **function** GREEDYCOST(x, demand)
2: Calculate positive hits: $posHits$
3: Calculate negative hits: $negHits$
4: $demand = demand - x$
5: $hitsDiff = posHits - negHits$
6: $cost = e^{-hitsDiff}$
7: **return** $(demand, cost)$
8: **end function**

Above given *Greedy Cost function* will return updated demand and cost of candidate element. In order to understand this *function* let's consider following example:

- $demand = [1, 2, 3, -2, 4, -5, -6]$
- $x = [1, 0, 0, 1, 1, 0, 0]$

Positive Hits is number of times x decreases from positive values of *demand*, while *Negative Hits* is number of times x decreases from negative val-

ues of *demand*. So in above given example, $posHits = 2$ and $negHits = 1$. Since our goal is preferebly to decrease positive values of leftover demand, $hitsDiff$ is used. After that, exponential function is applied to $hitsDiff$, because we want to exponentially decrease/increase cost in case of wrong/right candidate element and choose the maximum cost. Since we have minimization problem, we invert exponential function, i.e. $\max e^{hitsDiff} = \min e^{-hitsDiff}$.

3.2 BRKGA

Our approach is to sort the demand list according to a given chromosome. Then we will try to satisfy the demand of given hour iteratively. In addition to that, we will try to satisfy the given hour "neighbors" (as explained later). After satisfying all the demands, we legalize the solution. We make sure all problem's constrains are satisfied. In case of success, we return solution table and the total number of working nurses.

3.2.1 Decoder

The following algorithm receives set of data that contains the problem constrains. For simplification we will use the notations from the Problem Statement above.

function CONSTRUCT(data, chromosome)
  Sort the Demand list according to the chromosome.
  $Used \leftarrow \emptyset$
  for all $dIdx \in Demand$ **do**
    if $Demand_{dIdx} <= 0$ **then**
      continue
    end if
    for $nrs \in Nurses$ **do**
      if $can\_work(.)$ **then**
        $Used \leftarrow Used \cup \{nrs\}$
        Assign nrs to $hour_{dIdx}$
        $--\, Demand_{dIdx}$
        for $distance \in \{1 : maxConsec | stepsof 2\}$ **do**
          ▷ Assign nrs to neighbors of $hour_{dIdx}$ with interval of 2
          if $hour_{dIdx-distances} \in \{hour_{start}, hour_{end}\}$
            and $Demand_{dIdx-distances} \geq \frac{Demand_{dIdx}}{2}$
            and $can\_work(.)$ **then**
              Assign nrs to $hour_{dIdx-distances}$
              $--\, Demand_{dIdx-distances}$
          end if
          if $hour_{dIdx+distances} \in \{hour_{start}, hour_{end}\}$
            and $Demand_{dIdx+distances} \geq \frac{Demand_{dIdx}}{2}$
            and $can\_work(.)$ **then**
              Assign nrs to $hour_{dIdx+distances}$
              $--\, Demand_{dIdx+distances}$

26: **end if**
27: **end for** ▷ End of "neighbors" loop
28: **end if**
29: **if** $Demand_{dIdx} \leq 0$ **then** *break*
30: **end for**
31: ▷ Finish iterate over all the available nurses
32: **end for** ▷ iterate over demand list
33: **if** $(\exists hour | Demand_{hour} \geq 0)$ **then**
34: **return** $\{\emptyset\}, MaxInt$
35: **if** $(Legalize(solution))$ ▷ Check if all the constrains are satisfied
36: **return** $solution, SizeOf(Used)$
37: **return** $\{\emptyset\}, MaxInt$
38: **end function**

In words, our decoder will iterate over the demand list and satisfy the demand of *hour* with the first available *nurse*. When it will find a *nurse* that can work at *hour*, it will try to assign the *nurse* to work also in "neighborhood" of *hour* as we can see in L:13. After iterating all the demand list, our decoder will try to *Legalize* the the suggestion solution. In case of success, it will return the *solution* and the $fitness$ (number of working nurses).

The sorting phase in L:2 is straight-forward, the index of the largest chromosome will be the index of the first *hour* to satisfy in the demand list. The index of the second largest chromosome will be the index of the second *hour* to satisfy in the demand list etc.

"neighborhood" concept example L:13: assume we satisfy the demand of $hour = 5$, $nurse_h = [0, 1, 0, 0, 0, 1, 0, 0, 1, 0]$, and $maxConsec = 4$. The neighbors hours that we will try to assign *nurse* to are between $hour_{start} = 1$ and $hour_{end} = 8$ with interval of 2 around $hour = 5$, therefore $neighborhood = [2, 4, 6]$. In addition, the demand in each of the hours will be larger than half of $demand_{hour}$. In our example, $demand_2 > demand_5/2$, $demand_4 > demand_5/2$ and $demand_6 > demand_5/2$.

3.2.2 Check maxConsec, maxHours, maxPresence constrains

can_work(.) function will return $True$ if nurse can work in the given *hour*, $False$ otherwise.

```
1: function CAN_WORK(nurse_h, hour, constrains)
2:     nurse_hour ← 1
3:     if (nurse work more than maxConsec hours) then return False
4:     if (nurse_end − nurse_start > maxPresence) then return False
5:     if ∑ nurse_h > maxHours OR
6:         nurse_end − nurse_start > maxHours + 0.2 ∗ maxPresence then
7:         return False
8:     if (two maxConsec appears with two hours rest in between) then
9:         return False
10:    return True
11: end function
```

can_work(.) Function will make sure that we won't assign *nurse* to work in *hour* that will break *maxConsec*, *maxPresence* Or *maxHours* constrains. Max Consecutive/Presence hours are straight-forwards constrains L:3 & L:4. Max hours is a trick constrain, If we will just enforce it by summing the number of working hours we might get too many unfeasible solutions. Therefore, we choose to limit the range of possible work hours, it is similar to *maxPresence*, but, in same cases it will be larger than *maxPresence* and vise-verse. In L:5 we can see the actual heuristic.

Last, we want to avoid assignment to *hour* that we will not be able to legalize later. For example, given $nurse_h$: [0, 1, 1, 0, 0, 0, 1, 0], $maxConsec = 2$. assigning *nurse* to work in $hour = 5$ will not break the consecutive working hours constrain, $nurse_h$: [0, 1, 1, 0, 0, 1, 1, 0]. This assignment is illegal as it has two consecutive rest hours, Trying to legalize it will cause a violation of *maxConsec* constrain.

3.2.3 Legalize

Legalize function will return $True$ if the nurse working hours satisfied all constrains, $False$ otherwise. The function has wrapper that iterate over all working nurses. If one of the nurses fail to satisfied the constrains, it will stop iterate and return $False$.

```
1: function LEGALIZE(nurse_h, constrains)
2:     for all i ∈ {nurse_start : nurse_end} do ▷ Enforce 1 hour rest constrain.
3:         if (nurse_i + nurse_{i+1} == 0) then
4:             Assign nurse to work in i or i + 1, base on can_work(.)
5:     end for
6:     if (∑ nurse_h < minHours) then
7:         Assign nurse more hours
8:     if (minHours and Rest constrains satisfied) then
9:         return True
10:    return False
```

11: **end function**

Legalize function has two main goals. **First**, make sure there are no two consecutive rest hours. In L:2 we make sure that it won't happen by assign the nurse to work in case she/he rest for two consecutive hours. In our implementation we extended it to spot three consecutive rest hour, in such case, we assigned the nurse to work in the middle hour. See examples in Table.1. **Second**, verify that a nurse will work more than $minHours$, we do so by adding working hours using $can\_work(.)$ function.
MaxConsec, MaxHours and MaxPresent constrains were checked as part of the assigning in $can\_work(.)$

Input	Output	comment
[0, 1, 1, 0, 0, 1, 1, 0]	[0, 1, 1, 1, 0, 1, 1, 0]	
[1, 1, 0, 0, 0, 1, 1, 0]	[1, 1, 1, 0, 0, 1, 1, 0]	Output can't be legalized (maxhour=5, maxConsec=3) solution in next example
[1, 1, 0, 0, 0, 1, 1, 0]	[1, 1, 0, 1, 0, 1, 1, 0]	Nurse must be assigned in the middle

nHours: 8, *minHours:* 2 *maxHours:* 5 *maxConsec:* 3

Table 1: Consecutive rest elimination

4 Comparative Results

Before starting comparing results of metaheuristics and ILP, let's find the most optimal α parameter. And we must note that we allow overflow of available nurses. In other words, since metaheuristics are not as optimal as ILP, we may use up to $2 \times nNurses$ number of nurses. Finding optimal α parameter should be done on following pretty big instance size, to see tentative difference:

- *nNurses=1800*
- *nHours=24*
- *minHours=6*
- *maxHours=18*
- *maxConsec=7*
- *maxPresence=24*
- *demand=[964, 650, 966, 1021, 824, 387, 828, 952, 611, 468, 403, 561, 862, 597, 1098, 855, 918, 1016, 897, 356, 615, 670, 826, 349]*

From Figure 1 we can clearly see that the most optimal value of α parameter is 0.35 both timewise and objective function valuewise. Which means, we will use this α parameter for all future executions.

Figure 1: GRASP: Alpha vs Time & Objective Function Value

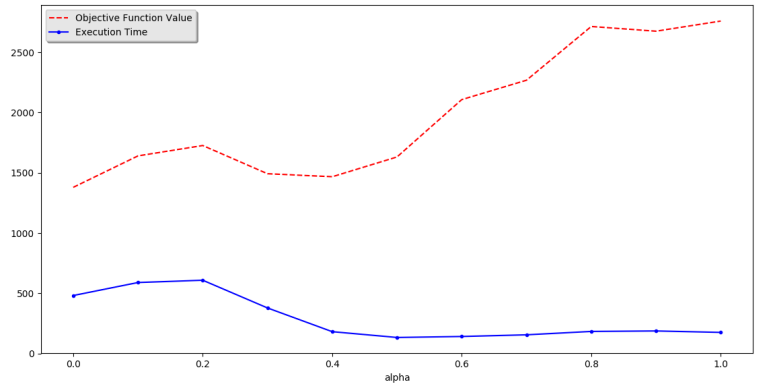


Figure 2: Objective Function Value Comparison

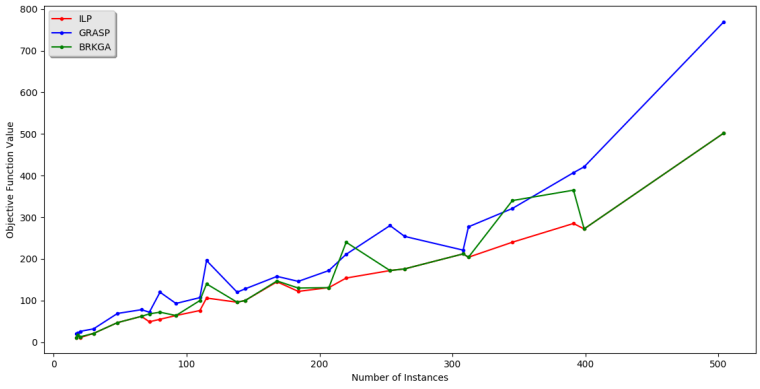
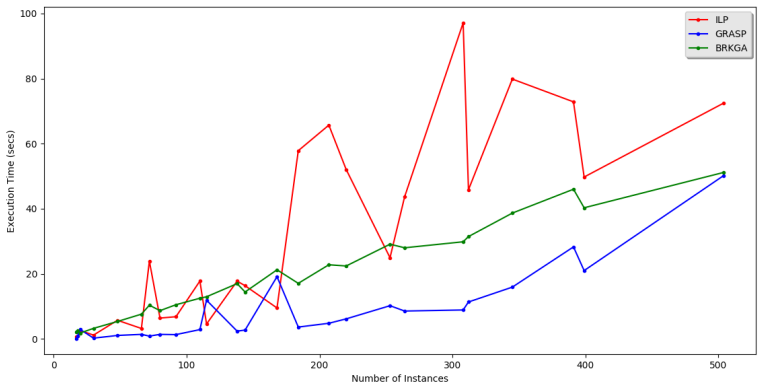


Figure 3: Execution Time Comparison



After running *ILP,GRASP* and *BRKGA* on small, medium and big instance sizes we can draw conclusions and compare the methods.

From Figure 2 we clearly see that *GRASP* doesn't perform bad on small and medium and some big size instances objective function valuewise. The values are not very far away from the optimal found by *ILP*. While *BRKGA* is performing slightly better than *GRASP*. Some of solutions are very close or even equal to optimal solution found by *ILP*, while others are almost as same as solutions found by *GRASP*.

However from Figure 3 we can observe the huge difference in execution time until finding a solution. *GRASP* is way faster than both *ILP* and *BRKGA* for medium and big instance sizes. So we pretty much get a trade-off between time and efficiency. If we can afford to run software for a longer time, then it's better to use *ILP* rather then *GRASP*. However, *BRKGA* seems the most optimal out of 3 methods, since its execution time is somewhere in the middle. And objective function value is close to global optimal found by *ILP*. In order to understand how *BRKGA* performs, let's run it on the very big data instance given beginning of Section 4. The result is depicted in Figure 4. We can see that result is converging to optimal found by *ILP* - 1098 pretty fast.

Figure 4: Execution of BRKGA