Report of the AMMM Projet

Mathieu Chiavassa, Anthony Nixon
Master MIRI Data Science, FIB, BarcelonaTech, UPC

January 9, 2018

Contents

‘

1 Problem statement

The problem we have to solve is to find an optimal daily schedule for a given number of nurses in an hospital. The optimal workload is given by the hospital for all the hours of the day. The nurses schedule must satisfiy a list of constraints.
To model the problem, we introduce the following input parameters :

- $numNurses$: Number of nurses available
- $hours$: Total number of hours in one working day
- $demand$: Vector representing the number of nurses who are supposed to work at each hours
- $minHours$: Minimal number of working hours for a given nurse
- $maxHours$: Maximal number of working hours for a given nurse
- $maxConsec$: Maximal allowed number of consecutive working hours for a given nurse
- $maxPresence$: Maximal number of hours the nurses can spend at the hospital for a given nurse

With these parameters, we can introduce the different constraints :

C1 : For each hour h, at least, $demand[h]$ nurses should be working

C2 : Each nurse should work at least $minHours$ hours

C3 : Each nurse should work at most $maxHours$ hours

C4 : Each nurse should work at most $maxConsec$ consecutive hours

C5 : No nurse can stay at the hospital for more than $maxPresence$ hours

C6 : No nurse can rest for more than one consecutive hour

The project consists in solving this problem using three different optimization methods: the linear model with the OPL software and the two metaheuristic methods BRKGA and GRASP.

2 Linear model for optimization in OPL

In order to solve this optimization problem in OPL we used the aforementioned parameters but we also introduce the following ones :

- range N : Range between 1 and $numNurses$
- range H : Range between 1 and $hours$
- works[numNurses][hours] : Boolean matrix of size $numNurses * hours$.
  Tells for each nurse if he/she is working at a given hour,i.e, if $works[i][j] = 1$ then the nurse i is working during the hour j
- worksBefore[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- worksAfter[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- rests[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- used[numNurses] : Boolean vector telling if a nurse is working during the day (value 1) or not (value 0).

The objective function to be minimized is therefore

$$\sum_{i=1}^{n} used[i].$$

The solution must satisfy the aforementioned constraints C1 to C6 we expressed in the following form to be used by the OPL code:

C1 : $\forall h, \sum_{i=1}^{n} works[i][h] \geq demand[h]$
For each hour, the sum of nurses working should be greater or equal than the number of nurses needed.

C2 : $\forall n, \sum_{i=1}^{h} works[n][i] \geq minHours * used[n]$
For each nurse, the sum of working hours in her schedule should be greater or equal than the minimum number of working hours.

C3 : $\forall n, \sum_{i=1}^{h} works[n][i] \leq maxHours * used[n]$
For each nurse, the sum of working hours in her schedule should be lower or equal than the maximum number of working hours.

C4 : $\forall n, \forall j \in [1; hours - maxConsec], \sum_{i=j}^{j+maxConsec} works[n][i] \leq maxConsec * used[n]$

C5 : $\forall n$, $\forall h$ if $h \leq hours - maxPresence$:
$worksBefore[n][h] + worksAfter[n][h + maxPresence] \leq 1$

C6 : $\forall n$, $\forall h \leq hours - 1$:
$worksAfter[n][h] \geq worksAfter[n][h + 1]$
$worksBefore[n][h] \leq worksBefore[n][h + 1]$
$rests[n][h] + rests[n][h + 1] \leq 1$
$rests[n][h] = (1 - works[n][h]) - (1 - worksAfter[n][h]) - (1 - worksBefore[n][h])$

The full OPL code in written the appendix A.

2.1 Results for the OPL code

In this section, we are going to present 3 tests based on 3 different datasets, a small one (TEST1), a medium one (TEST2), and a big one (TEST3). The different datasets are in the OPL data file.

TEST1:
In this test we try to optimize the schedule of 30 nurses for a 9 hours working day, with the following demand workload :
`demand = [ 5 3 8 5 1 7 5 6 2 ];`
And the following nurses parameters:
`numNurses = 30; hours = 9; minHours = 3; maxHours = 6; maxConsec = 7; maxPresence = 8;`

On this small dataset, OPL took less than 1 second to find the optimal solution. The objective function is equal to 8, which is the max of the demand vector. It means that not a single nurse is useless, and the solution is optimal.

TEST2:
In this test we try to optimize the schedule of 200 nurses for a 24 hours working day, with the following demand workload :
`demand = [ 53 24 33 40 70 12 33 55 66 12 30 22 55 77 88 22 34 55 22 55 23 22 11 12 ];`
And the following nurses parameters:
`numNurses = 200; hours = 24; minHours = 6; maxHours = 12; maxConsec = 6; maxPresence = 18;`

OPL took only 3 minutes to solve this problem. But the objective function value is 108, which is higher than the max of the demand vector, so we are not sure if we can assume that this is the optimal solution.

TEST3:
In this test we try to optimize the schedule of 1800 nurses for a 24 hours working day,

with the following demand workload :

```
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];
```

And the following nurses parameters:

```
numNurses = 1800; hours = 24; minHours = 6; maxHours = 18; maxConsec = 7;
maxPresence = 24;
```

Since it is a very huge dataset, OPL took 1 hour and 13 minutes to optimize this problem. But the result is excellent, the objective function is equal to 1098 nurses, which is the max of the demand vector. So we can be sure that this solution is the optimal solution.

3 The BRKGA method

In this section, we use a Biased Random Key Genetic Algorithm (BRKGA), to solve the nurses scheduling problem. This method belongs to the genetic algorithms family and has been introduced by Bean (1994) and developed later for solving large combinatorial optimization problems. The basic principle is to mimic the evolution of individuals in a population submitted to natural selection. Basically, for a given criteria, called *fitness*, the best individuals are selected and are crossed with the other individuals of the population. Generations after generations, the individuals that optimize the fitness are selected.
We based our work in this section on the paper of José Fernando Goncalves and Mauricio G. C. Resende, *Biased Random-Key Genetic Algorithm for Combinatorial Optimization* and on the python code provided by profesor M. Ruiz Ramirez.

Each individual of the population is represented by a *chromosome* and corresponds to a candidate for the solution we are looking for. The chromosome consists of a given number of *genes* that takes a real value in the interval $[0, 1]$. The gene are used to compute the fitness of the individual which is correlated to the problem we want to optimize. To solve a minimization problem such the nurse scheduling described in section 1, the BRKGA used the following steps:

1 define the size p of the population, and the number of generations

2 initialize randomly the genes of the p individuals

3 decode the genes to compute the fitness of each individual

4 select the best individuals, *elite*, and generate the next generation chromosomes, see figure 1

5 go to step 3 until the number of generations is not reached

6 decode the best fitness chromosome of the last generation

If the problem and the parameters have been chosen correctly, the individual with the best fitness in the last generation is a good candidate for the optimum of our problem.

The tricky part of the BRKGA algorithms is the decoder procedure and a huge number of research papers deal with this problem. If the chromosome is composed of N genes, the decoder must map the hypercube space $[0, 1]^N$ to the solution space. A suited fitness has also to be defined o select the best candidate in the solution space.

We read many papers to find the best way to define such procedure for our nurse scheduling problem. Nevertheless we do not succeed to apply the proposed ideas due to the constraints we have on the schedules. We then proposed our own decoder technique, which as we will see, works well in some cases but has some weakness we will comment later on.

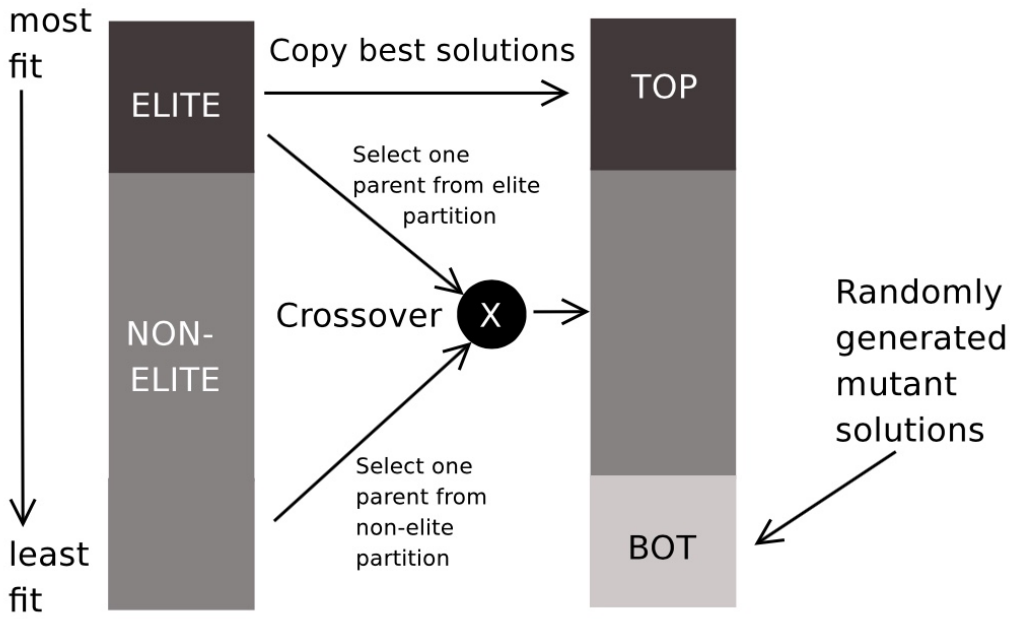

Figure 1: Schematic representation for transition to generation k to $k+1$. (Reproduced from J.F. Goncalves paper). A percentage of the best fitted individuals, called elite is directly reproduced in the next generation. A percentage of the new generation is composed by crossover between elite and non-elite individuals. Crossover is obtain by randomly exchanging genes of the parents. The third part of the new generation is composed by mutants whose genes are randomly generated. This allows to explore a larger space of solutions, and prevents (theoretically) to be trapped in a local optimum.

3.1 Our decoder and fitness procedures

The decoder procedure we propose will map the chromosome to the daily schedule for all the nurses, represented by the boolean array $works$ of size $numNnurse \times hours$ introduced in section 2. The chromosome length is therefore $numNurse \times hours$. The mapping from floating genes value to boolean ones is simply done by the transform procedure:

Procedure transform

```
for all the genes of the population
  if gene_value ≤ 1-minHours/hours then solution_value=0
  else solution_value=1
```

With this procedure, the probability of working for a nurse depends on the percentage she has to work during the day. If the ratio is small, then the transform procedure will produce more 0 values (not work). On the contrary more values 1 will be produced. (This improves significantly the results in our experiments comparing to use the fixed value 0.5).

Once we have the daily schedule, we will check if the schedule satisfies all the constraints of the problem described in section 1 and define a corresponding value of the fitness. First for a given nurse, its schedule is verified:

Procedure checkschedule

```
test=0
  check all the constraints from C2 to C6 defined in section 1
  for each non satisfied constraint do test + = 1
return test
```

The way we check in practice the constraints is detailled in the procedure `checkschedule` in Annexe B. The value of test will be used for the fitness.

We then compare the workload of the day with the theoretical demand specified by the user, in order to check if the constraint C1 is satisfied:

Procedure compareworkload

```
error=0
for all the hours h of the day
 compute the number of working nurses:  workload(h)
 if workload(h) > demand(h) then error + = workload(h)-demand(h)
 if workload(h) < demand(h) then error + = 10*|workload(h)-demand(h)|
return error
```

For this error we penalize 10 times more the fact that the workload is less than the demand. When the error is larger than the demand, the optimal soltuion is not reached, but the solution is acceptable in practice. If the workload is exactly aquals to the demand, the error is 0.

We can then define the complete decode procedure

Procedure decode

```
transform
for all the individuals in the population
  fitness=0
  for all the nurses
    test=checkschedule
    fitness += 10.*test/numNurses
  error=compareworkload
  fitness += error/hours
```

With this definition of the fitness, we have the following properties:

- if all the nurses schedules are ok and if the workload is exactly the same than the demand, then the fitness is equal to 0
- when a nurse schedule is not satisfactory, the fitness is increased by an amount of $10 * test/numNurses$. It means that the penalization depends on the number of constraints that are not satisfied. The factor 10 has been found from the experiments and is necessary to gives more weight to the schedule part of the fitness than to the error one. In practice it means that obtaining correct schedules is important! We also normalize by the number of nurses to have a criteria mostly independant of the data.
- when the workload is not optimal, the fitness is increased depending of the error normalized by the number of hours in the day. Due to the definition of the error function, the fitness is higly penalized for the hours where the workload is less than the demand.

Since the python BRKGA procedure proposed by in this project is written to minimize the fitness, we use directly this decoding procedure in the full BRKGA python code.

The python scripts of all our procedures are given in Appendix B and are written in the file DECODER_DUMMY.py. The specific data are in the DATA_DUMMY file. We hope our contribution is not too dummy!

3.2 Results for BRKGA method

We present three tests in this section, a simple one (TEST1), a medium one (TEST2) and un more complex (TEST3). The data corresponding to these tests are in the file

DATA_DUMMY. All the tests are performed on a MacBook with Intel Core i7 at 2.7 GHz.

TEST1:
We first applied the algorithm for a simple test with 3 nurses and 6 hours to validate the method. For this test, the following data are used in the genetic algorithm:
`'chromosomeLength': 18, 'inheritanceProb': 0.7, 'numIndividuals': 20, 'eliteProp': 0.1, 'maxNumGen': 100, 'mutantProp': 0.3`
The results for the best individual are the following and the evolution of the fitness is plotted on figure 2.

```
('Nurse', 0, 'Test Schedule =', 0)
('Nurse', 1, 'Test Schedule =', 0)
('Nurse', 2, 'Test Schedule =', 0)
('Comp. Workload', [0, 2, 1, 1, 0, 1])
('Dema. Workload', [0, 2, 1, 1, 0, 1])
```

After less than 50 generations, the fitness is 0 since all the constraints have been satis-

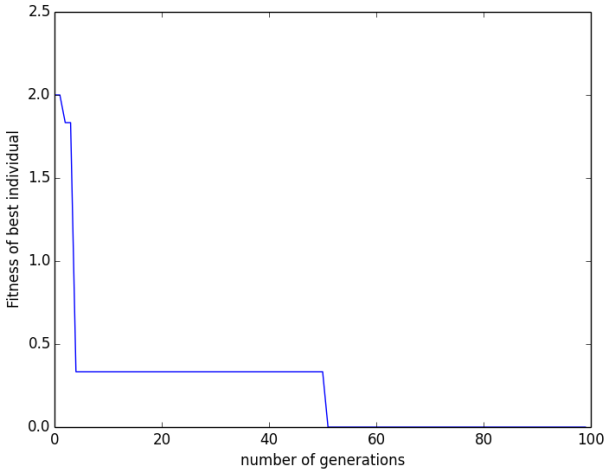

Figure 2: Evolution of the fitness for the TEST1.

fied (Test Schedule =0) and the workload is equal to the demand. An optimal solution is then obtained. The running time is neglectible and much less than 1 second.

TEST2:
This test optimize the schedule of 10 nurses for a 12 hours working day. After some run, we obtain a perfect scheduling using the following parameters:
`'chromosomeLength': 120, 'inheritanceProb': 0.7, 'numIndividuals': 100, 'eliteProp': 0.1, 'maxNumGen': 500, 'mutantProp': 0.3`
We obtain the optimal solution: all the schedules of the nurses satisfy the constraints,

the demand is exactly satisfied. The corresponding fitness is represented on figure 3 and reaches the 0 value after 60 generations. `('Nurse', 0, 'Test Schedule =', 0)`

```
('Nurse', 1, 'Test Schedule =', 0)
('Nurse', 2, 'Test Schedule =', 0)
('Nurse', 3, 'Test Schedule =', 0)
('Nurse', 4, 'Test Schedule =', 0)
('Nurse', 5, 'Test Schedule =', 0)
('Nurse', 6, 'Test Schedule =', 0)
('Nurse', 7, 'Test Schedule =', 0)
('Nurse', 8, 'Test Schedule =', 0)
('Nurse', 9, 'Test Schedule =', 0)
('Comp.  Workload', [0, 2, 4, 5, 2, 7, 6, 4, 3, 3, 1, 1])
('Dema.  Workload', [0, 2, 4, 5, 2, 7, 6, 4, 3, 3, 1, 1])
```

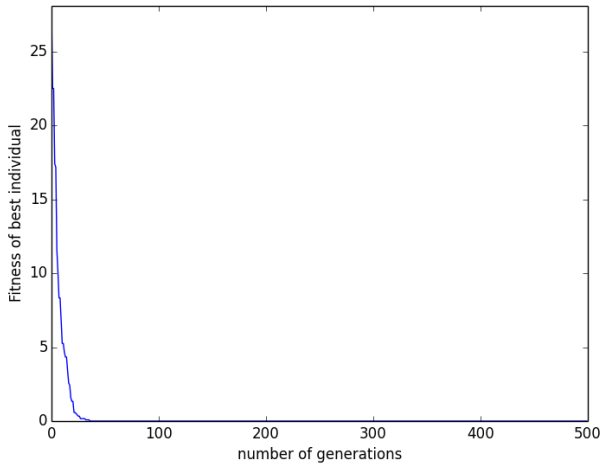

Figure 3: Evolution of the fitness for the TEST2.

This solution is obtain in less than 7 secondes. For all the runs we made for this TEST2, we mostly obtain a zero fitness solution in less than 1000 generations. In the case where the fitness is not 0, the schedules are always correct but the workload is overestimated since too much nurses are working comparing to the demand. This corresponds to an acceptable solution, but not the optimal one.

TEST3:
In this test we try to optimize the schedule of 25 nurses for a 18 hours working day where the demand workload is :

```
Dema.  Workload', [9, 8, 8, 10, 11, 12, 9, 10, 16, 11, 10, 14, 13, 13, 9,
```

15, 12, 10],
and the nurses parameters:
'nNurses': 25, 'nHours': 18, 'minHours':4, 'maxHours':8, 'maxPresence':10, 'maxConsec':5
In that case, we have 450 genes for each individual of the population.

One typical exemple is reported here, with the following AG parameters:
'chromosomeLength': 450, 'inheritanceProb': 0.7, 'numIndividuals': 150, 'eliteProp': 0.1, 'maxNumGen': 2000, 'mutantProp': 0.3
The result is not so bad since only 2 nurses have an incorrect schedule and for 2 hours the workload is not equal to the demand. The fitness evolution is reported on figure 4 a). As observed on figure 4 b), increasing the number of generations do not improve the result.

We perform a huge number of numerical experiments exploring the parameters space to try to obtain the optimal solution. Unfortunately, we never obtain this optimal solution!
We increase the number of mutants, and the probability of crossover, to avoid falling in a local minima but the results were not significantly better. Adding more individuals in the population do not always results in better results, but of course significantly increase the cpu time!

Sometimes we obtain a perfect schedule for all the 25 nurses but in that case the workload is far from the demand, and do not improves with the number of generations. We tried to find an definition of the fitness with different amount of penalization coming from schedule and workload errors, but once again no optimal solution has been found.

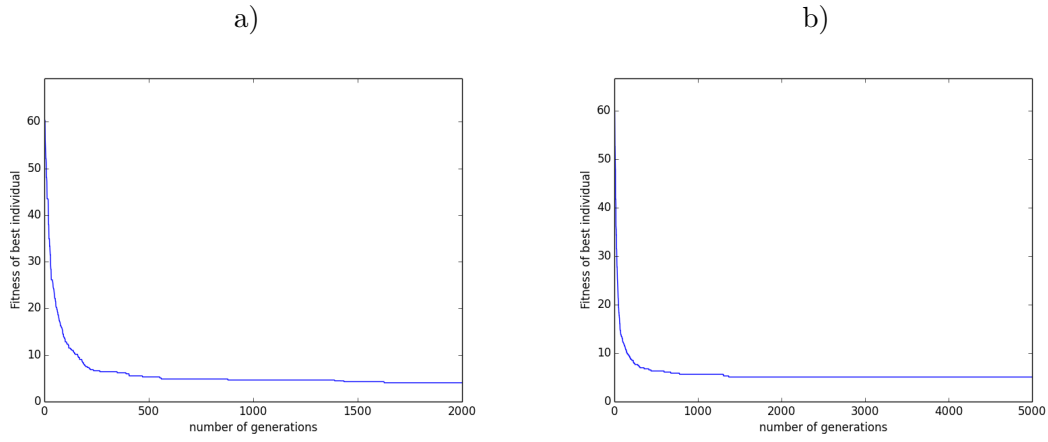

Figure 4: Evolution of the fitness for the TEST3. a) with 2000 iterations (170 sec), b) with 5000 iterations (7 min)

3.3 Conclusion for BRKGA method

We found this method very interesting since it is based on very simple concepts to optimize a complex problem. Our version gives most of the time the optimal solution for problems with less than about 350 genes. Nevertheless, we are a little bit disappointed since we do not obtain optimal results for larger problems, despite many efforts...
Some improvements could be made, adding also a mutation procedure of the chromosomes, as we have seen in many AG papers. It will may be help to go out of the local minima by modifying randomly the genes of the individuals.
But the best improvements would be to define a decoding procedure that gives directly a correct schedule for all the nurses. In that case, only the workload would be optimized in a very smaller space. Unfortunately, we did not found this procedure, but we will be very interesting if it exists!

4 The GRASP method

The greedy randomized adaptive search procedures method (GRASP) has a constuction phase and a local search phase. Iteration through these phases obtains a good, but not necessarily optimal, solution by creating feasible solutions then refining them during each iteration. In our project the following algorithms were implemented:

Procedure construct(α, C = candidate_set)

```
possible_solution = ∅
solved = False
Initialize demand vector D⃗
for k = 1,2,...,numNurses do
  s_min = min{g(t, D⃗) | t ∈ C}
  s_max = max{g(t, D⃗) | t ∈ C}
  RCL = {s ∈ C | g(s, D⃗) ≤ s_min + α * (s_max − s_min)}
  select random s from RCL
  possible_solution = possible_solution ∪ s
  Subtract s from D⃗
  if max(D⃗) is 0 do
    solved = True
    break
  end if
end for
if solved is True do
  return possible_solution
else infeasible
```

The constuct function builds up a feasible solution (one where all the demand is satisfied by a subset of available nurses with valid schedules) by adding one nurse at a time until all the demand is satisfied or the pool

of nurses is exhausted. A solution is a matrix of dimension numNurses x numHours where each row is a binary vector representing a nurse schedule where 1 is a worked hour and 0 is unworked. Many GRASP implementations will update the candidate list after each addition, however, in our problem nurses are independent of each other and the schedule of one nurse has no effect on the constraints of the next nurse. We only need to update the demand. The candidate list of possible schedules is preconstructed against the constraints and passed in as an argument.

To decide which schedule to assign each nurse added to the solution we score how each schedule in the candidate list will affect the demand vector via our **greedy function g()** described below. After scores are assigned to candidates we create a restricted candidate list which creates a sub-list of best candidates based on the **RCL Equation**: $s_{min} + \alpha * (s_{max} - s_{min})$. α is a value [0-1] which defines a percentage of best scored candidates. A value of 0 means it's most greedy and will always pick the best, a value of 1 means highest randomness and any candidate might be added to solution at that stage. The point of the RCL is to introduce an element of randomness to the constructed solution to get new local search minimums.

Procedure g(schedule, $\vec{D}$ = demand)

```
1 score = 0
2 maxd = max(D)
3 for k = 1,2,...,numNurses do
4   Δ = D[k] - schedule[k]
5   if Δ not 0 do
6    score = score + exp^(delta/maxd)
7   end if
8 end for
9 return score
```

Our **GREEDY function g()** uses the exponential function to accumulate a score on how effective the schedule at satisfying demand. If the schedule was a perfect match (Δ = 0) no penalty score would be added. We chose exponential because it accumulates penalties for both failing to staff a demand hour and for overstaffing a demand hour, the case when Δ of a given hour is 1 or -1 respectively. Lastly, we normalized the function because of buffer limitations in runtime.

Procedure local_search(x = feasible_solution)

```
1 neighbor_set = {y ∈ N(x) | f(y) < f(x)}
2 best_neighbor = x
3 while neighbor_set is ≠ ∅ do
4   select random n from neighbor_set
```

```
5   best_neighbor = n
6   neighbor_set = {y ∈ N(best_neighbor) | f(y) < f(best_neighbor)}
7 end while
8 return best_neighbor
```

Local search refines the initial feasible solution but generating a set of nearby solutions defined by the neighbor function N() (described below), which improve the objective function f() (described below). From this set the algorithm descends to a local min by choosing a random solution and then repeating the process (finding it's neighbor set and comparing their f() values), until we are left with a single element which has no better neighbors.

Procedure f(solution)

```
1 return number of nurses in solution (count rows in solution)
```

Objective function f() returns the number of nurses used in a solution by returning the number of rows, each of which represent one nurse working in the solution.

Procedure N(x = solution)

```
1 neighbor_set = ∅
2 Initialize demand vector D⃗
3 nurses_working = {(t_1, t_2, ..., t_n) | ∀j ∈ {1, ..., nHours}, t_j = Σ_{i=1}^{numNurses} x_ij}
4 demand_Δ⃗ = nurses_working⃗ - D⃗
5 for each row in solution do
6   Initialize neighbor_sol to solution
7   row_effect⃗ = demand_Δ⃗ − row⃗
8   negative_count = number of negatives in row_effect⃗
9   if negative_count is 0 do
10    remove row from neighbor_sol
11    add neighbor_sol to neighbor_set
12  end if
13 end for
14 return neighbor_set
```

The neighbor function N() returns a set of all neighbors to the input solution. It defines a neighbor to be any solution that still satisfies demand after removing a nurse from the original solution. To implement this we tracked demand surpluses from the original solution in the $\vec{demand\_\Delta}$ variable and then simulated how this overall demand remained satisfied after each possible row removal via $\vec{row\_effect}$. A negative value in this vector indicates that there is demand that is unsatisfied.

A Appendix : OPL script

OPL optimization code :

```
int numNurses = ...;
int hours = ...;
range N = 1..numNurses;
range H = 1..hours;

int demand [h in H]= ...;
int minHours = ...;
int maxHours = ...;
int maxConsec = ...;

int maxPresence = ...;

dvar boolean works[n in N][h in H]; // this set of variable should suffice
for A). Tells whether nurse n works at hour h
dvar boolean worksBefore[n in N][h in H]; // no nurse can rest more than
one consec 1/3
dvar boolean worksAfter[n in N][h in H]; // 2/3
dvar boolean rests[n in N][h in H]; // 3/3
dvar boolean used[n in N];



   minimize sum(n in N) used[n]; // do not change this for A)
subject to {

// the number of provided nurses is greater or equal to the demand
forall(h in H)
sum(n in N) works[n][h] >= demand[h];

// Each nurse should work at least minHours hours.
forall(n in N)
sum (h in H) works[n][h] >= minHours*used[n];

// Each nurse should work at most maxHours hours.
forall(n in N)
sum (h in H) works[n][h] <= maxHours*used[n];

// Each nurse should work at most maxConsec consecutive hours.
forall(n in N)
forall(i in 1..(hours-maxConsec))
```

```
sum(j in i..(i+maxConsec)) works[n][j] <= maxConsec*used[n];

// No nurse can stay at the hospital for more than max Presence
// hours (e.g.  if maxP resence 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).
forall(n in N)
forall (h in H: h <= hours-maxPresence)
worksBefore[n][h] + worksAfter[n][h+maxPresence] <= 1;

   forall(n in N)
forall (h in H: h <= hours-1){
worksAfter[n][h] >= worksAfter[n][h+1]; // legal:  11111110, illegal:  11111010
worksBefore[n][h] <= worksBefore[n][h+1]; // legal:  00011111, illegal:  00111110
rests[n][h] + rests[n][h+1] <= 1;
// legal:  00010100, illegal:  00110010
}
forall(n in N)
forall (h in H)
rests[n][h] == (1-works[n][h]) - (1-worksAfter[n][h]) - (1-worksBefore[n][h]);
}

execute { // Should not be changed.  Assumes that variables works[n][h] are
used.
for (var n in N) {
write("Nurse ");
if (n < 10) write(" ");
write(n + " works:  ");
var minHour = -1;
var maxHour = -1;
var totalHours = 0;
for (var h in H) {
if (works[n][h] == 1) {
totalHours++;
write(" W");
if (minHour == -1) minHour = h;
maxHour = h;
}
else write(" .");
}
if (minHour != -1) write(" Presence:  " + (maxHour - minHour +1));
else write(" Presence:  0")
writeln ("(TOTAL " + totalHours + "
}
```

```
writeln("");
write("Demand:  ");

for (h in H) {
if (demand[h] < 10) write(" ");
write(" " + demand[h]);
}
writeln("");
write("Assigned:  ");
for (h in H) {
var total = 0;
for (n in N)
if (works[n][h] == 1) total = total+1;
if (total < 10) write(" ");
write(" " + total);
}
}
```

B Appendix : BRKGA script

Here are the python procedures we write corresponding to the BRKGA method. We apologize for the fact that the number of hours of the day, named $hours$ in the document is called $nHours$ in the code.

Transform the gene value onto a boolean value corresponding to the daily schedule

```
def transform(population):
  for i in range(len(population)):
      for j in range(len(population[i]['chr'])):
        if population[i]['chr'][j] <= 1.-1.*data['minHours']/data['nHours']:
          population[i]['solution'][j]=0
        else:  population[i]['solution'][j]=1
  return population
```

Check if the constraints are satisfied by the nurse schedule:

```
def checkschedule(currentnurse):
test=0
sumhours=sum(currentnurse)
if sumhours < data['minHours'] and sumhours != 0:  # Constraint minH
```

```
      test+=1
if sumhours > data['maxHours']:  #Constraint maxH
      test+=1
if sumhours != 0:  #Constraints rests and maxPresence
      i=0
      while currentnurse[i]==0:
        i=i+1
      imin=i
      i=len(currentnurse)-1
      while currentnurse[i] == 0:
        i = i - 1
      imax=i
      for i in range(imin, imax):
        if currentnurse[i]==0 and currentnurse[i+1]==0:  #Constraints rests
          test+=1
      if imax-imin+1 > data['maxPresence']:  #Constraints maxPresence
        test+=1
      compteur=0
      for i in range(imin, imax+1):  #Constraints maxConsec
        if currentnurse[i]==1:
          compteur+=1
        else:
          if compteur > data['maxConsec']:
            test+=1
          compteur = 0
      if compteur > data['maxConsec']:
        test+=1
return test
```

Comparison between workload and given demand:

```
def compareworkload(workload):
      error=0
      for i in range(0,data['nHours']):
        if workload[i]-data['demand'][i] > 0:
          error+=(1.0*workload[i]-1.0*data['demand'][i])/data['nHours']
        if workload[i]-data['demand'][i] < 0:
          error += 10.0*abs((workload[i] - data['demand'][i])) / data['nHours']
return error
```

The decoder procedure:

```
def decode(population,data):
      population=transform(population)
      for x in range(len(population)):
        population[x]['fitness']=0
        workload=[0]*data['nHours']
        for i in range(0, data['nNurses']):
          currentnurse = []
          for j in range(i*data['nHours'], (i+1)*data['nHours']):
            currentnurse.append(population[x]['solution'][j])
          test=checkschedule(currentnurse)
          #penalize fitness if bad schedule
          population[x]['fitness'] += 10.0*test/data['nNurses']
          for k in range(0, data['nHours']):
            workload[k]+=currentnurse[k]
        error=compareworkload(workload).
        # penalize the fitness if workload error
        population[x]['fitness'] += error.
return(population)
```

We also define a procedure **infoBestIndividual** that allow us to analyze the final solution in term of schedule checking and workload demand. The script is also in the DECODER_DUMMY.py file.