Algorithmic Methods for Mathematical Models: COURSE PROJECT

Ricard Gardella, Sofia B. Reichl

January 9, 2018

Abstract

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 $demand_h$ 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 inimize the number of nurses required and satisfy all the afore mentioned constraints.
In order to deal with the hospital schedule, we present and evaluate a formal model of integer linear programming (**ILP**) formulation. We also make use of two well-known and powerful metaheuristic models to help solve the problem, these are, a greedy randomized adaptive search procedure (**GRASP**) and a biased randomkey genetic algorithm (**BRKGA**). To evaluate both heuristics, we carry out a set of experiments using both methodologies and assess their respective performances.

Linear Model

To solve our linear model we used these parameters:

- *NumNurses:* Number of nurses.
- *Hours:* Number of hours.
- *Demand:* Demand of nurses each hour.
- *MinHours:* Minimum number of hours that should be worked.
- *MaxHours:* Maximum number of hours that should be worked.
- *MaxConsec:* Maximum number of consecutive hours that can be worked.
- *MaxPresence:* Maximum number of hours that a nurse that works should be in the hospital.

We also used a **set of nurses** (of size *NumNurses*) and a **set of hours** (*Hours*' size).

Our decision variables were:

- *works*: This will be an integer matrix of size NumNurses x Hours. As it is integer it can only takes values of 1 or 0. The cell will be 1 if the nurse works at that hour and 0 if not.
- *nurseworks*: This will be an integer array of size NumNurses. As it is integer it can only takes values of 1 or 0. The cell will be 1 if the nurse works or 0 if not. This variable is the one that will be used in the objective function.
- *worksBefore*: This will be an integer matrix of size NumNurses x Hours. As it is integer it can only takes values of 1 or 0. The cell will be 1 if the nurse had been working before or 0 if not.
- *worksAfter*: This will be an integer matrix of size NumNurses x Hours. As it is integer it can only takes values of 1 or 0. The cell will be 1 if the nurse will work after or 0 if not.
- *rests*: This will be an integer matrix of size NumNurses x Hours. As it is integer it can only takes values of 1 or 0. The cell will be 1 if the nurse rests at that hour or 0 if not.

The linear model we are going to solve using the previous parameters and variables is going to be the next one:

$$\min \sum_{n}^{N} nurseworks[n] \tag{1}$$

s.t:

$$\sum_{n}^{N} works[n,h] \geq demand[h] \qquad \forall h \in H \tag{2}$$

$$\sum_{h}^{H} works[n,h] \geq minHours * nurseworks[n] \qquad \forall n \in N \tag{3}$$

$$\sum_{h}^{H} works[n,h] \leq maxHours * nurseworks[n] \qquad \forall n \in N \tag{4}$$

$$hours * works[n][h] + \sum_{k=h+MaxPresence}^{hours} works[n,k] \leq hours \qquad \forall n \in N, h \in H \tag{5}$$

$$\sum_{k}^{h+MaxConsec} works[n,k] < MaxConsec \qquad \forall n \in N, h \in H \tag{6}$$

The objective function (1) sum up all the nurses that will work, because our main objective is to minimize this number.

The (2) constraint is related to the demand of nurses per hour. For each hour h at least *demand* nurses should be working.

(3),(4) Are constraints related to the number of hours that should be worked by any nurse (between *minHours* and *maxHours*). In these constraints we are using also the variable *nurseworks* in order to give it the binary value (if the nurses works or not).

(5) This constraint is related to the maximum presence of a nurse in the hospital. Taking the total number of hours and the binary value of works and then summing the value with the summation of all the hours going from the actual hour + maxpresence to the total number of hours. This operation have to be less than the total number of hours.

(6) The sixth constraint is about the number of consecutive hours that can be worked by any nurse. We are summing all the hours worked by each nurse (in groups of *MaxConsec* length) and make it lower than the maximum number of consecutive hours.

Metha-Heuristics

GRASP

The GRASP procedure is an iterative two phase meta-heuristic method based on a multi-start randomized search technique with a proven effectiveness in solving hard combinatorial optimization problems

Createsolution()
This part of the GRASP Heuristic creates a random solution, with a given demand. Using the demand given, function sums either 0 to the value of demand extra value in order to make the solution really random. In this part of the code, the constraints of rest, maxhours and minhours are implemented. The last part of the code, add random hours in the case the constraint of minimum hours is not passed. The idea is to create a nurse with the maximum hours possible and then, go to the next nurse and try to create a nurse with the maximum possible hours in order to minimize the total nurse used.

FeasibleFunction() This function tells if a given solution is feasible or not. We check all the constraints, except the rest constraint that is imposed in the creation of the solution. If all the constraints are full filled, the solution if correct.

ConstructiveGRASP() This is the code for the constructive phase of the algorithm. The constructive phase will iterate until it finds the number of solutions specified at the variable numbersolutions. In the end, the constructive phase will return only the best found solution based on the lower cost and the lower number of nurses

getCost() In this function the cost of the GRASP solution is computed. We penalize the extra hours worked in the solution, so, if it's more efficient, lower cost it will have. Also we multiply the final cost for the total number of nurses, so, if we have less nurses, less cost we will have. In the end, we want to minimize the total number of nurses.

localSearch() The local search function takes a feasible solution and tries to reduce the total number of nurse, deleting the unnecessary nurses of the solution and then, returning the solution which is still feasible.

BRKGA

The **BRKGA** is a genetic algorithm which is made between a solution and an individual in a population. Each individual is represented as an array of ng genes, called chromosome, and each gene can take a value, called an allele, in the real interval [0,1]. Each chromosome encodes a solution of the problem and a fitness level, that is related to the objective function value. In our problem we used the chromosome to order all the sets that we were studying to make the schedule: the hours and the nurses.
A set of individuals (population) evolves over a number of generations. At each generation, individuals of the current generation are selected to mate and produce offspring, making up the next generation. In BRKGA, individuals of the population are classified into two sets: the elite set p, with those individuals with the best fitness values, and the non-elite set. Elite individuals are copied unchanged from one generation to the next, thus keeping track of good solutions. The majority of new individuals are generated by crossover, that is, by combining two elements, one elite and another non-elite, selected at random. Finally, to escape from local optima a small number of mutant individuals (pm, randomly generated) are introduced at each generation to complete a population. A deterministic algorithm, named decoder, transforms any input chromosome into a feasible solution of the optimization problem and computes its fitness value. This decoder associates a solution of combinatorial optimization problem for which an objective value or fitness can be computed.

Practice of BRKGA:
As we have said in our case we used the chromosome in order to sort our elements: nurses and hours. We have these in the *decoderorder()* function. This function has several steps:

1. We sort the nurses and the hours.
2. In the first iteration we only take a small subset of nurses.
3. We take the selected nurses (with the *ordernurses()* function) by the chromosome order to test if we can make the whole schedule with only these subset of nurses. That is what *myfunction()* does.
4. In *myfunction()* we iterate all the hours, while the minhours worked by a nurse are not recovered, and see if the nurse is available for those hours. That is made thanks to the *availablenurse()* function.
5. In the *availablenurse()* function we study the constraints related to the maximum number of hours that can be worked by a nurse, the number of consecutive hours, the presence hours in the hospital and the rest hours. If that nurse is able to work that hour it returns true, if not it returns false.
6. After that if the minimum hours are not recovered by that nurse we stop taking her into account in our current schedule.

7. If the demand of that hour hasnt been fulfilled, we add another nurse (with the *getnewnurse()* function) and iterate again.

If its necessary we can see all the functions in detail in the annex of the project. We obtain (from the decoder) a set of solutions with its fitness. Once we have them, the classification of the solutions begin. At the end, what we have is the best solution of our algorithm, which has been compared to others and has the best fitness.

In the practice, after testing, we found out that this algorithm doesn't work as well as we wanted. This happens because of the complexity of our function, which could be much simpler. Our function doesn't work well with big sets and hours with high demand.

Comparative Results

ILP vs. Metaheuristics

In this section we will compare the results from the integer linear model with the ones from the metaheuristics, in order to see the efectivity of our implementations. First of all we will start with a very simple example:

```
nNurses=12
hours=10
minHours=2
maxHours=5
maxConsec=8
maxPresence=7
demand=c(2, 2, 1, 1, 1, 3, 2, 4, 1, 2)
```

We had solved this example with the three options and these were the results:

ILP

- Solving time: 0.0058S
- Objective value: 5 nurses
- Solution:

```
works =
 [[0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 1 1 1 0 1]
  [0 0 0 0 0 1 1 1 0 1]
  [0 0 0 0 0 1 0 1 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 1 1 1 1 1]
  [1 1 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [1 1 1 1 1 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]];
```

GRASP

- Solving time: 0.0366s
- Objective value: 10 nurses
- Solution:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
1	0	1	1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	1	0	1	0
3	0	0	0	0	0	1	1	1	0	1
4	1	0	1	0	1	0	1	1	0	0
5	1	1	0	1	0	1	0	1	0	0
6	1	0	1	0	1	1	0	1	0	0
7	0	0	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	1	0	1

BRKGA

- Solving time: 4.2s
- Objective value: 6 nurses
- Solution:

```
solution:
[[0 0 0 0 0 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 1 1 1 0 1]
 [1 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 1 1 1]
 [1 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 1 1 1 1 0 1 0 0]]
('fitness:', 50.0)
time 0:00:04.204000
```

In this simple example we can see some of the differences of our algorithms. The ILP is the best approach for solving some linear models problems, because they are able to find the best solution. The meta-heuristics try to find that solution but they didn't know if it's the best one. Thanks to the example we can see that the ILP and the BRKGA had found the same solution, but the BRKGA was slower than the ILP. If we compare them to the GRASP, it seems that this algorithm doesn't work as well as we wanted with such a small sample. If we focus on the results in terms of solving time and objective value we can say that the best solution is given by the ILP.

We also found interesting to compare how a larger problem instances works with our implementations:

```
nNurses=1100
nHours=24
minHours=6
maxHours=18
maxConsec=7
maxPresence=24
demand=c(130, 400, 400, 764, 300, 387, 624, 430, 611, 468, 403, 561, 700, 597, 430, 855, 300, 230, 300, 356, 232, 670, 230, 349)
```

In this case we will only use the ILP and the GRASP algorithms to solve it. These are the results:

	ILP	**GRASP**
Objective:	855 nurses	1.069 nurses
Solving time:	11:09:60s	30:52 s

We can obviously see that the ILP had found a better solution than the GRASP, but we have to take into account the solving time. The GRASP is so much faster than our integer linear problem implementation. Maybe the GRASP could be able to find a better solution with more time.
In conclusion, if we want a good solution, without taking into account the solving time, the best implementation is the integer linear model. When we use GRASP

or BRKGA we have to be aware that they are meta-heuristics, that means that they are able to find a good solution which cannot be the best one.

GRASP vs. BRKGA

In this section we will compare both heuristics in terms of solving time and quality of the results.
The first set of parameters we had used was this:

```
nNurses=30
nHours=24
minHours=2
maxHours=12
maxConsec=8
maxPresence=18
demand=c(4, 2, 3, 1, 2, 2, 6, 2, 2, 1, 4, 1, 2, 2, 2, 5, 2, 1, 1, 1, 1, 3, 1, 2)
```

The second one is this:

```
nNurses=42
nHours=24
minHours=2
maxHours=18
maxConsec=8
maxPresence=24
demand=c(11,7,5,1,4,6,9,10,3,7,11,2,8,6,5,6,6,2,1,3,2,6,5,1)
```

The last one with more nurses:

```
nNurses=73
hours=24
minHours=2
maxHours=18
maxConsec=8
maxPresence=24
demand=c(13,17,7,12,4,14,12,10,4,6,7,4,15,12,9,13,12,11,1,3,2,2,2,3)
```

These are the results:

Algorithms	Comparison	param1	param2	param3
GRASP	Solving time	11.65s	0.0668s	0.113s
	Objective value	22 nurses	33 nurses	48 nurses
BRKGA	Solving time	01.49s	05.26s	16.14s
	Objective value	13 nurses	33 nurses	44 nurses

Thanks to the table we can see that BRKGA has better solutions than GRASP in terms of the objective function. In the case of the solving time is clear that it is the BIG drawback of our BRKGA implementation. The BRKGA meta-heuristic is faster in the first example because, as we had seen our implementation start working with less nurses than the all available ones. Thus, the solution is found faster. The demand that we are using in these examples is very slow, so, it seems that it is easy to fulfill. To sum up, it's obvious that if we are looking for a fast and a good solver we can use our GRASP implementation, because it's very effective.

Test Environment

For doing the results of the GRASP algorithm this Environment has been used:

- Operating System: Mac OS X 10.13.2
- Processor: 2,4 GHz Intel Core i7
- Memory: 8 GB 1600 MHz DDR3
- Hard Drive: APPLE SSD SM0512F - 256Gb SSD
- Dedicated Graphics: NVIDIA GeForce GT 650M 1GB
- Integrated Graphics: Intel HD Graphics 4000 1536 MB
- Programming Language: R

For the BRKGA algorithm we used this environment:

- Operating System: Lenovo YOGA 710
- Processor: 2,7 GHz Intel Core i7
- Memory: 8 GB
- Programming Language: Python

Appendix

0.1 GRASP

Create Solution

```
createSolution <- function(works, demand, maxHours,minHours, hours, nNurses, maxConsec){
    auxDemand <- rep(0,hours)
    nHour = 0
    consecutive = 0
    for(h in 1 : hours) {auxDemand [h] = demand[h] + round(runif(1,0,demand[h]))} #Random values to the solution
    for (nurse in 1 : nNurses){
      nHour = 0
      consecutive = 0
      for(hour in 1 : hours){
        if(auxDemand[hour] >= 1 & nHour < maxHours){  #If a nurse have 8 hours, can't assign more than that. #Demand satisfier
          works[nurse,hour]= ifelse(runif(1,0,1)>=0.4,1,0)#Assign a random value between 1 and 0.
          #Forcign resting 1h at max
          if(consecutive == maxConsec)
          {
            works[nurse,hour]= 0
          }
          if(hour > 2)
          {
            if(works[nurse,hour-1]== 0 & works[nurse,hour-2]== 1) {
              works[nurse,hour] = 1
              consecutive = consecutive + 1
            }
          }
          if(works[nurse,hour] == 1)
          {
            auxDemand[hour] = auxDemand[hour] - 1  #Substract to auxdemand
            nHour = nHour + 1
            consecutive = consecutive + 1
          }
        }
        else{
          if(nHour < maxHours & hour > 2)
            {
              if(works[nurse,hour-1]== 0 & works[nurse,hour-2]== 1)
                {
                works[nurse,hour] = 1
                auxDemand[hour] = auxDemand[hour] - 1
                nHour = nHour + 1
                consecutive = consecutive + 1
                }
            }
        }
        if(works[nurse,hour] == 0) consecutive = 0
      }
      #Si entra aqui añadira m´ás horas de las necesarias, por tanto, la soluci´ón ser´á m´ás cara.
      if(sum(works[nurse,]) < minHours & sum(works[nurse,])>0)#Si tenemos menos horas que el minimo, pero tenemos alguna, añadimos más
        {
        for(hour in 1 : hours){
          if(sum(works[nurse,])< minHours)
          {
            if(works[nurse,hour] != 1)
            {
              works[nurse,hour]= ifelse(runif(1,0,1)>=0.5,1,0)#Assign a random value between 1 and 0
            }
          }
        }
      }
    }
    return(works)
  }
```

Feasible Function

```
feasibleFunction <- function(hours, maxHours, minHours, nNurses, maxConsec,maxPresence,demand,works){
  for (nurse in 1 : nNurses){
    if(sum(works[nurse,])>0){
      if(sum(works[nurse,]) > maxHours || sum(works[nurse,]) < minHours)
        {
        return(FALSE)#Sum hours nurse between min and max hours
        }
      for (hour in 1 : hours){ #not sure
        if(hour == 1)
        {
          FirstHour = 0 #firstHour that the nurse work
          LastHour = 0 #lasthour that the nurse work
          consecutive = 0
          worksBefore= 0
          rest = 0
        }
        #Maxpresence
        if(works[nurse,hour]==1 && FirstHour == 0)
        {
          FirstHour = hour
          LastHour = hour
        }
        if(works[nurse,hour]==1) LastHour = hour
        if(LastHour - FirstHour > maxPresence) {
          return(FALSE)  }
        #maxconsec
        if (works[nurse,hour]== 1) {
          consecutive = consecutive + works[nurse,hour]
        }else {
          consecutive = 0
        }
        if(consecutive > maxConsec) {
          return (FALSE) }
        #MaxRest
        if(sum(works[,hour]) < demand[hour])
        {
          return(FALSE)
        }
      }
    }
  }
  for (hour in 1 : hours){
    #Demand is fulfilled.
    if(sum(works[,hour]) < demand[hour])
      {
      print("Demand")

      return (FALSE)
      }
  }
  return(TRUE)
}
```

ConstructiveGRASP()

```
auxCostSol = 0
bestSol = 0
costbestsol = 0
numberNursesBestSol = nNurses
start_time <- Sys.time()
while(number_solutions != 0)
{
  solution <- matrix(ncol = hours , nrow = nNurses ,data = 0) #Nurses x hour
  solution = createSolution(works = solution,demand = demand,maxHours = maxHours,minHours = minHours, hours = hours,nNurses=nNurses,maxConsec = maxConsec)
  if(feasibleFunction(works = solution,hours = hours,maxHours = maxHours, minHours = minHours,nNurses = nNurses,maxConsec = maxConsec,maxPresence = maxPresence,demand =
  {
    solution = localsearch(solution = solution,demand = demand,maxHours = maxHours,minHours = minHours,maxPresence = maxPresence,maxConsec = maxConsec, hours = hours)
    auxCostSol = getCost(works =solution,hours = hours,maxHours = maxHours,nNurses = nrow(solution),maxConsec = maxConsec,maxPresence = maxPresence,demand = demand)
    if(costbestsol == 0)
    {
      costbestsol = auxCostSol
      bestSol = solution
      numberNursesBestSol = nrow(solution)
    }
    if(auxCostSol < costbestsol & nrow(solution) <= numberNursesBestSol)
    {
      costbestsol = auxCostSol
      bestSol = solution
      numberNursesBestSol = nrow(solution)
    }
    number_solutions = number_solutions -1
  }
}
print(Sys.time() - start_time)
print("Nurses used:")
print(nrow(solution))
print("Cost after localsearch")
return(costbestsol)
```

getCost()

```
getCost <- function(hours, maxHours, nNurses, maxConsec,maxPresence,demand,works){ #Something is missing
  costHoraNurse=1
  costNurse= 1000
  cost = nNurses
  for (nurse in 1 : nNurses){
    cost = cost + costNurse
    for(hour in 1 : hours){
    cost = cost + costHoraNurse * works[nurse,hour] #Works nurse will be 1 or 0.
    }
  }
  for(hour in 1 : hours)cost = cost + sum(works[,hour]) - demand[hour] #Adding cost if there is extra hours worked
  cost = cost * nNurses #More nurses = more cost
  return(cost)
}
```

localSearch()

```
localsearch <- function(solution, demand,maxHours,minHours,maxPresence,maxConsec)
{
  auxSolution = solution
  for(nurse in 1: nrow(auxSolution))
  {
    suma_hores<-vector()
    for(hour in 1 : hours)
      suma_hores[hour] <- sum(solution[,hour])
    extrahours = demand - suma_hores
    delete = TRUE
    deleteNurses = max(extrahours)
    if (deleteNurses != 0 ) solution = solution[-c(1:abs(deleteNurses)), ]
  }
  return(solution)
}
```

0.2 BRKGA

decoderorder()

```
def decoder_order(data, chromosome):

    def my_function(nurse_order_used, nurse_remain):


    N = [0]*(int(data["numNurses"]))
    N_notused = [0]*int(data["numNurses"]*0.4)
    N_used = [0]*(int(data["numNurses"]-int(len(N_notused))))   #vector de nurses utilitzades

    used = N

    chr_nurse = chromosome[0:len(N)]
    nurse_order = sorted(range(len(N)), key = lambda k: chr_nurse[k])
    nurse_order_used = N_used
    nurse_not_used = N_notused
    nurse_order_used, nurse_remain = order_nurses(nurse_order)

    schedule_aux, fitness_aux = my_function(nurse_order_used, nurse_remain)


    return schedule_aux, fitness_aux
```

ordernurses()

```
def order_nurses(nurse_order):
    ## Triem quines nurses utilitzar tenint en compte l'ordre del cromosoma
    aux_nurse_order = nurse_order
    aux = []
    for nurse in aux_nurse_order:
        if nurse == min(aux_nurse_order):
            aux += [aux_nurse_order.index(nurse)]
            aux_nurse_order[aux_nurse_order.index(nurse)] = sys.maxint
    if len(aux)==1:
        aux += aux
    return aux, aux_nurse_order
```

myfunction()

```
def my_function(nurse_order_used, nurse_remain):
    aux = nurse_order_used
    schedule = np.zeros(int(data["numNurses"])*int(data["hours"]),dtype=np.int).reshape((int(data["numNurses"]),int(data["hours"])))
    H = [0]*int(len(schedule[0][:]))       # vector d'hores
    demand = map(float, data["demand"][:])
    auxdemand = demand

    chr_hour = chromosome[len(N_used):len(chromosome)]
    hour_order = sorted(range(len(H)), key = lambda k: chr_hour[k])

    for i in aux:
        nurse_worked = np.sum(schedule[i,:])
        nursedemand = auxdemand
        while (int(data["minHours"]) > nurse_worked):# or (used[i] == 1):
            for j in hour_order:
                if auxdemand[j] >= 0: #mentre hi ha demanda a la hora j
                    if available_nurse(schedule.copy(),i,j,data):
                        schedule[i][j] = 1 #assignem nurse i a la hora j
                        used[i] = 1
                        nursedemand[j] -= 1 # restem demanda de la hora j
                        auxdemand_ok = nursedemand
            nurse_worked = np.sum(schedule[i,:])

            if int(data["minHours"]) > nurse_worked:
                schedule[i,:] = 0 # reassignem a la nurse i a l'hora j que acabem d'assignar el 0 --> al final no pot treballar perque no cobreix el minim d'hores.
                used[i] = 0
                nursedemand = auxdemand
                break
            else: auxdemand = auxdemand_ok
            if int(data["maxHours"]) == nurse_worked:
                auxdemand = auxdemand_ok
                break
    fitness = (int(sum(used))/int(data["numNurses"]))*100   # proporcio de nurses utilitzades
```

```
    positive_list = []
    for x in auxdemand:
        if x > 0:
            positive_list += [x]
    val = False
    for z in range(len(schedule[0,:])):
        demandz  = map(float, data["demand"][:])
        if(int(sum(schedule[:,z])) < int(demandz[z])):
            val = True

    if val == True:
        if int(len(nurse_order_used)) < int(len(N)) :
            value, aux_nurse_order = get_new_nurse(nurse_order_used, nurse_remain)
            nurse_order_used += [value]
            my_function(nurse_order_used, aux_nurse_order)
        else:
            final_schedule = None
            final_fitness = sys.maxint
            return final_schedule, final_fitness
    else:
        for k in range(len(schedule[0,:])):
            demandk  = map(float, data["demand"][:])
            if(int(sum(schedule[:,k])) < int(demandk[k])):
        final_schedule = schedule
        final_fitness = fitness
        return final_schedule, final_fitness
```

availablenurse()

```
def available_nurse(sch,nurse,hour,data):
    dummySchedule = sch
    dummySchedule[nurse][hour] = 1

    # Parameters of data
    minHours = int(data["minHours"])
    maxHours = int(data["maxHours"])
    maxConsec = int(data["maxConsec"])
    maxPresence = int(data["maxPresence"])

    if np.sum(dummySchedule[nurse][:]) == 1:
        return True

    # MaxHours --> Si supera les hores maximes no pot
    if sum(dummySchedule[nurse][:]) > maxHours:
        #print "maxhours fails"
        return False

    # PRESENCE
    first_hour_worked = np.nonzero(dummySchedule[nurse][:])[0][0]
    last_hour_worked = max(np.nonzero(dummySchedule[nurse][:])[0])
    if (last_hour_worked - first_hour_worked + 1) > maxPresence:
        return False


    # CONSECUTIVE
    worked = 0
    for j in range(int(data['hours'])):
        if(dummySchedule[nurse, j] == 1):
            worked += 1
            if worked > maxConsec:
                return False
        else:
            worked = 0

    # REST
    first_hour_worked = np.nonzero(dummySchedule[nurse][:])[0][0]
    last_hour_worked = max(np.nonzero(dummySchedule[nurse][:])[0])
    lastHour = 0;
    for j in range(first_hour_worked, last_hour_worked+1):
        currentHour = int(dummySchedule[nurse][j])
        if(currentHour == 0 and lastHour == 0):
            return False
        lastHour = currentHour

    # COMPLEIX REQUISITS --> ASSIGNAR
    return True
```

getnewnurse()

```
def get_new_nurse(nurse_order, aux_nurse_order):
    for nurse in aux_nurse_order:
        if nurse == min(aux_nurse_order):
            nurse_number = aux_nurse_order.index(nurse)
            aux_nurse_order[nurse_number] = sys.maxint
            return nurse_number, aux_nurse_order
```