ALGORITHMIC METHODS FOR MATHEMATICAL MODELS

COURSE PROJECT
REPORT

January 7, 2018

Kymry Burwell
Jerome Pasvantis

Contents

1 PROBLEM STATEMENT

The problem we are asked to solve is a typical scheduling problem. A hospital needs to create a schedule for their nurses. We are given a set of nurses and must determine at which hours each nurse works. Additionally, we are given a set of constraints that must be adhered to and a demand for the minimum number of nurses that must be working at a given hour.

In order to solve this problem we devise an integer linear model, as well as two meta-heuristics: GRASP and BRKGA. The linear model finds the optimal solution every time, but the time to find a solution is unbounded; it can take and unreasonable amount of time. In order to combat this deficiency, we created GRASP and BRKGA meta-heuristics. These don't always find an optimal solution, but typically end in a reasonable time, providing an adequate solution.

In the attached files you will find the code for the BRKGA, GRASP, and ILP models. The included readme.MD file provides execution instructions.

In the report we begin with a description of the ILP model and all constraints. We then provide pseudo-code and an explanation for the GRASP and BRKGA algorithms. Next, we analyze the performance of the models by inputing different data sets, increasing in size, and comparing the solutions and execution time. We then compare the BRKGA and GRASP algorithms in a number of different ways. Finally, we end with a conclusion that sums up our findings.

2 INTEGER LINEAR PROGRAMMING MODEL

Decision variables

$worksToday_n$: Boolean, true if a nurse works at some hour of the day
$working_{n,h}$: Boolean, true if a nurse works at the given hour
$worksBefore_{n,h}$: Boolean, true if a nurse worked before the given hour
$worksAfter_{n,h}$: Boolean, true if a nurse worked after the given hour

Objective function

Minimize the total number of nurses working.

$\forall n \in N$: minimize $\sum \text{worksToday}_n$

Constraints

(0) If a nurse works at least one hour, the boolean *worksToday* must be set to true indicating that they are working and therefore must meet all other constraints.

$\forall n \in N : worksToday_n * M \geq \sum_{h \in H} working_{n,h}$

(1) Demand for number of nurses each hour is met.

$\forall h \in H : \sum_{n \in N} working_{n,h} \geq demand_h$

(2) All working nurses must work at least minHours.

$\forall n \in N : \sum_{h \in H} working_{n,h} \geq minHours * worksToday_n$

(3) All working nurses can work no more than maxHours.

$\forall n \in N : \sum_{h \in H} working_{n,h} \leq maxHours * worksToday_n$

(4) All working nurses work at most maxConsec hours in a row without a break.

$\forall n \in N, \forall h \in H$ s.t. $h \leq hours - maxConsec : \sum_{c \in C} working_{n,h+c} \leq maxConsec$

(5) All working nurses can be at the hospital no longer than maxPresence (including working hours and breaks).

$\forall n \in N : (\sum_{h \in H} worksAfter_{n,h}) - (hours - \sum_{h \in H} worksBefore_{n,h}) \leq maxPresence - 2$

(6) No working nurses can rest for more than one hour at a time.

$\forall n \in N, \forall h \in H$ s.t. $h < hours : working_{n,h} + working_{n,h+1} \geq worksBefore_{n,h} + worksAfter_{n,h} - 1$

(Helper 1) Indicates if the nurse worked *before* the current hour (for constraints 5 and 6).

$\forall n \in N, \forall h \in H : worksBefore_{n,h} * M \geq \sum_{h_2 \in H : h_2 < h} working_{n,h_2}$

and

$\forall n \in N, \forall h \in H : worksBefore_{n,h} \leq \sum_{h_2 \in H : h_2 < h} working_{n,h_2}$

(Helper 2) Indicates if the nurse worked *after* the current hour (for constraints 5 and 6).

$\forall n \in N, \forall h \in H : worksAfter_{n,h} * M \geq \sum_{h_2 \in H : h_2 > h} working_{n,h_2}$

and

$\forall n \in N, \forall h \in H : worksAfter_{n,h} \leq \sum_{h_2 \in H : h_2 > h} working_{n,h_2}$

3 BRKGA

Brief Description

Our decoder intakes a chromosome with a length that is equal to the number of nurses. Each gene in the chromosome is a random number ranging from 0 to the number of hours the hospital is open, chosen uniformly at random with replacement. The random number signifies the first working hour assigned to the nurse. The following is an example.

[0, 12, 23, 5, 7, 19, 7, 16, 10, 1, 23, 3, 17, 22, 11, 18, 6, 18, 20]

The decoder starts with the first gene and successively works it's way through the chromosome until demand is met.

With each gene, it creates a new nurse and has that nurse begin working at the indicated hour. If the hour is a positive integer, the nurse works to the right, if negative they work to the left (e.g.: hour is 7, so nurse works hours 6,5, etc..). It then makes the nurse work as much as possible within the problem constraints before moving on to the next nurse/gene. During this process, if the nurse reaches an end hour (i.e. hour 0 or 23), they start working in the opposite direction (e.g.: hour in the gene is 3, so nurse works hours 3, 2, 1, and 0. After this, there are no more hours for the nurse to work, so the nurse begins working in the opposite direction. It works hours 4, 5, etc..). Nurses are forced to break for 1 hour after maxConsec is met.

Algorithm 1 BRKGA (decoder)

Input: data, chromosome **Output:** solution, fitness

```
1:  nurses ← empty set
2:  for gene in chromosome do
3:      if demandMet == true then
4:          solution ← computeSolution(nurses)
5:          fitness ← count(nurses)
6:          return solution, fitness
7:      nurses ← createNurse(id = iteration index)
8:      if gene == even then
9:          workingDirection ← forward
10:     else
11:         workingDirection ← backward
12:     startHour, currentHour ← gene
13:     while maxPresenc not met and maxHours not reached do
14:         assignHours
15:         if currentHour == 23 then
16:             currentHour ← startHour
17:             workingDirection ← backward
18:         else if currentHour == 0 then
19:             currentHour ← startHour
20:             workingDirection ← forward
```

The dataset and chromosome are the inputs. The solution and fitness are the outputs. The algorithm begins by creating an empty set to store the working nurses (line 1). It then iterates through all of the genes (line 2), ending when demand is met (line 3). During each loop, it creates a new nurse (line 7). It assigns a working direction based on the gene. If the gene is an even integer, the working direction is forward, while if the gene is an odd integer, the working direction is backward. (lines 8-11). Next, the startHour and CurrentHour are set to the gene value (line 12). While the max presence and max hours constraints have not been met, the nurse is assigned working hours in the specified direction. During this process, all problem constraints are adhered to. (**NOTE:** We did not include the constraints in the pseudocode because we think it makes it too confusing. The Python code is attached in case you would like to take a look). If hour 0 or hour 23 is reached, and the nurse is still allowed to work more hours, they begin working in the opposite direction from where they started. **Example:** A gene has a value of 3, so the nurse begins working at hour 3 and continues to work hours 2, 1 and 0. However, there is no hour -1, but they are still allowed to work. So, they proceed to work hours 4,5 etc. (lines 15-20). Once the nurse is assigned as many hours as possible, the algorithm moves on to the next nurse/gene. The decoder is essentially a greedy algorithm with a random input form the chromosome. It is truly greedy so the output is deterministic and gives the same output when given the same chromosome.

4 GRASP

The following three pseudocodes show how we implemented the constructive and local search phases of the GRASP.

Algorithm 2 GRASP (constructive)

Input: list of nurses, list of hours, α **Output:** solution, cost

```
Initialise candidate set C (all combinations of hours and nurses)
while C not empty do
    sort C by cost
    s_min ← first(C).cost
    s_max ← last(C).cost
    threshold ← s_min + α * (s_max − s_min)
    RCL ← all elements c ∈ C s.t. c.cost < threshold
    pick ← random element of RCL
    C ← C + tempRemoved
    updateCandidates(C, pick)
    if feasible solution found then return
    Remove unfeasible solutions (Constraints maxHours, maxPresence, maxConsec) from C
    tempRemove ← temporarily unfeasible solutions (Constraint consecRest)
    Remove temporarily unfeasible solutions from C
```

For the constructive phase we mostly follow the pseudocode from the GRASP paper. A candidate is a 3-tuple of nurse, hour and cost. The candidate set is initialized with every combination of nurses and hours. At the end of every iteration we check if we found a feasible solution, i.e. if every working nurse works at least minHours and the demand is met for every hour, the other constraints are ensured in the next step. Here we also remove all unfeasible candidates (i.e. the nurse's schedule would violate the maxHours, maxPresence or maxConsec constraints) from the candidate set. For the last constraint (no longer rest than one hour) we decided to temporarily remove all candidates that would create an unfeasible solution in the next iteration. We add those candidates again in the next iteration.

Algorithm 3 GRASP (updateCandidates)

Input: C, pick, list of nurses, list of hours **Output:** C

```
Add pick to list of nurses and list of hours
Remove pick from C
for c in C do
    if demand for the hour already met then
        c.cost + = X
    if nurse is working and works less than minHours then
        c.cost − = X
    if nurse not working yet then
        c.cost + = X
```

Everytime we choose a candidate in the constructive algorithm we call the function updateCandidates. In this function we add the candidate to the solution (i.e. adding it to the nurse and hour, from which we derive the solution later) and update the greedy cost for all the remaining candidates. We decided to penalize the use of new nurses (in order to minimize the number) and also the use of hours with already met demand (so that the other hours have priority). Nurses that already work should be cheaper if they didn't work for minHours yet (so they can also work in hours with met demand, if necessary).

Algorithm 4 GRASP (local)

Input: list of nurses, list of hours, cost **Output:** solution, cost

```
1:  for each nurse in nursesList do
2:      Feasible ← True
3:      for each hour nurse is working do
4:          if workingNurses - 1 < demand then
5:              Feasible ← False
6:          if Feasible then
7:              Remove nurse from solution
8:          if bestCost > cost -1 then
9:              bestCost ← cost -1
10:             solution ← Generate solution data from nurse objects
11:         local(nurses, hours, cost -1)
```

Our local search procedure takes our found solution as input and basically checks if one of the working nurses is maybe unnecessarily working. We check, if after removing one nurse, demand is still met for all hours. If so, we can remove the nurse from the solution (which decreases the cost by one). We do that over all nurses and do it recursively. So the neighborhood of each solution are all the solutions with one nurse removed.

5 ILP, GRASP AND BRKGA COMPARISON

In this section we compare the running time and best solution of the ILP, BRKGA, and GRASP algorithms over increasingly large and complex data sets. We started with a small data set of 20 nurses and low demand at each hour. For each new dataset we increased the number of nurses and the demand until we reached a total of 220 nurses in the final set. For GRASP we set the alpha to 0.4 and max iterations to 20. For BRKGA we set elite to 0.2, mutant to 0.2 and inheritance to 0.6. We chose these values because through experimentation we found them to be the best, on average. Datasets 1-4 were used for the comparison. Please see appendix A.

5.1 Solution Comparison

Figure 1 shows the best solution for each algorithm as the problem size increases. As we can see, when the problem size is small, there isn't much of a gap between the best solutions. However, as the problem size increases, the gap widens. In all cases the GRASP and BRKGA find solutions that aren't far off from the optimal ILP solution.

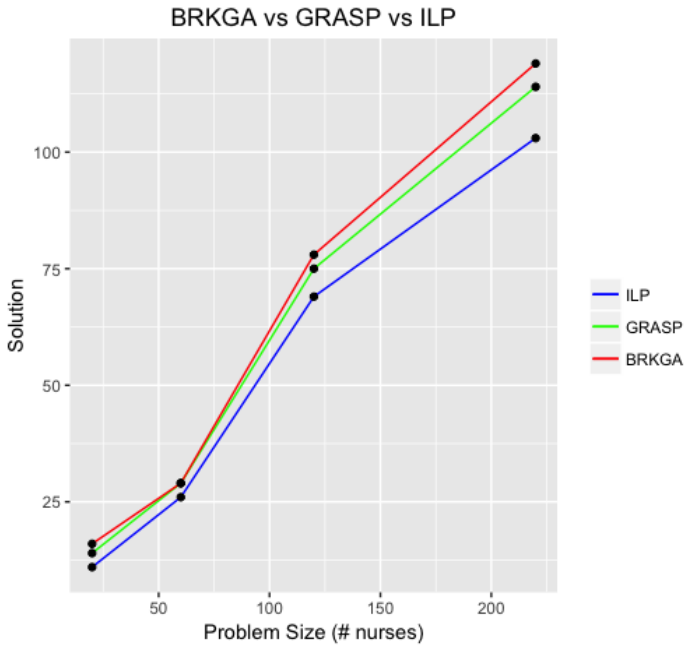

Figure 1: Solution Comparison

5.2 Running-time Comparison

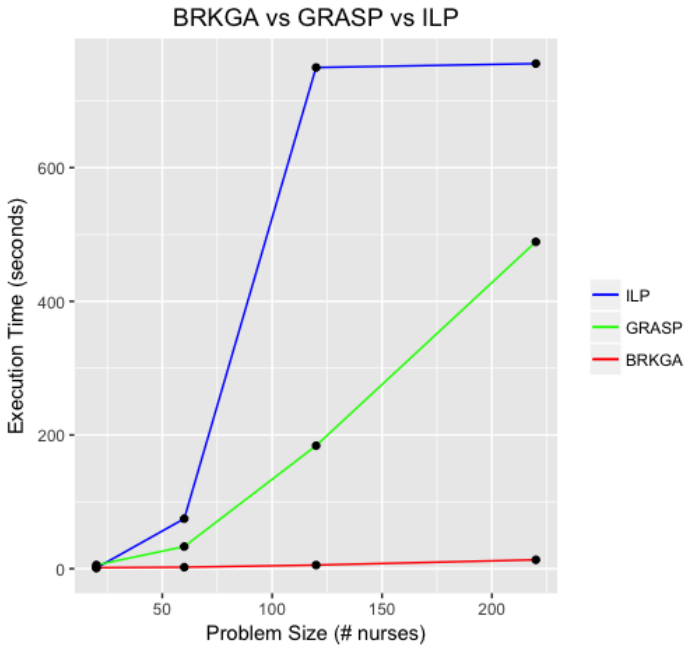

Figure 2: Running-time Comparison

Figure 2 plots the running-times of each algorithm over increasing problem sizes. We see that when the problem size is small, all three have roughly similar running times. However, as the problem size increases ILP takes drastically longer than both GRASP and BRKGA, and GRASP takes much longer than BRKGA.

6 GRASP AND BRKGA COMPARISON

6.1 Parameter Comparison

The left graph in figure 3 displays the performance of the GRASP algorithm with varying alpha. For this we used dataset 5 (Appendix A). We see that when alpha is small (< 0.7) the solution is quite good. It varies a bit, but much of that is likely due to the randomness of the algorithm. When alpha goes above 0.6, we see that the solutions get much worse.

The right graph in figure 3 displays the performance of the BRKGA algorithm with varying parameters. We tested different parameter combinations within the recommend limits supplied by J. Goncalves and M. Resend [1] (See figure 4 for parameter combinations). We chose to work within these limits as there would be far too many combinations to test otherwise. In order to prove that these are valid limits, we performed three tests with parameters far outside these limits (parameter combinations 7-9 in figure 3.). From figure 3, we see that the best parameter combinations have the mutant and elite in the 0.1 - 0.2 range and the inheritance in the 0.6 - 0.8 range.

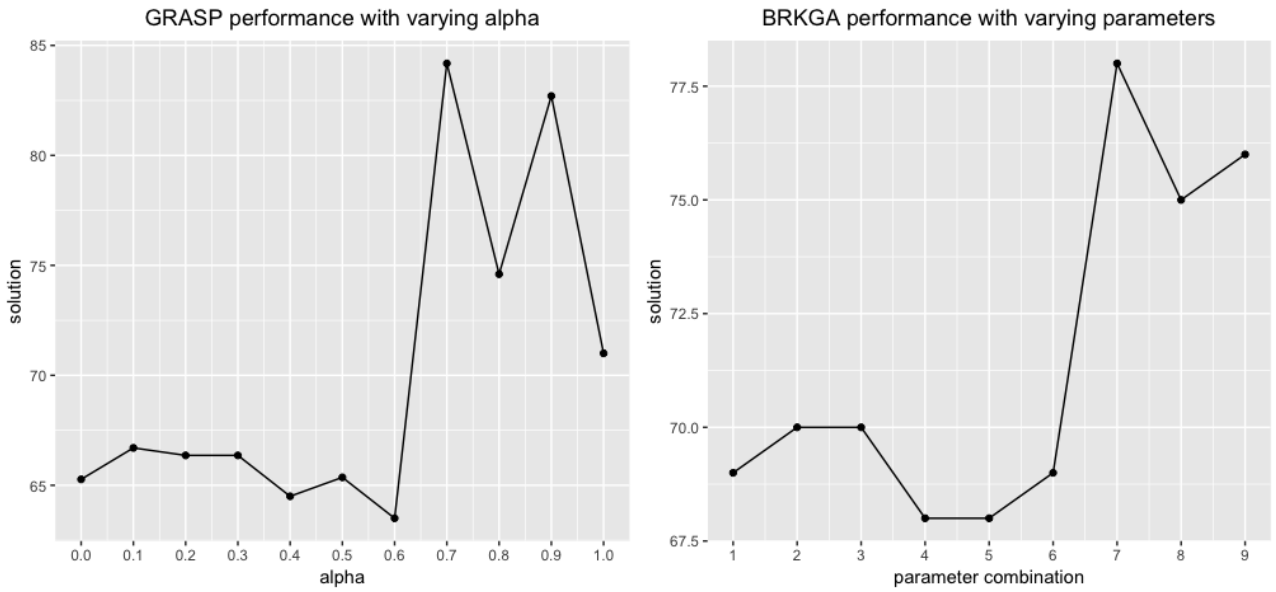

Figure 3: BRKGA and GRASP Parameter Combinations

BRKGA Parameters									
Parameter Combination	1	2	3	4	5	6	7	8	9
Elite	.1	25	.2	.1	.2	.25	.8	.4	.1
Mutant	.1	.3	.2	.2	.1	.1	.1	.4	.8
Inheritance	.5	.8	.6	.6	.8	.6	.9	.5	.1

Figure 4: BRKGA Parameters

6.2 BRKGA and GRASP Iterations comparison

For sections 6.2 and 6.3 we used the dataset 6 (Appendix 1). The left graph in figure 5 plots the solution of our GRASP algorithm against the current iteration. We can see that as that the solution is steady for the first 10 iterations, then changes drastically in just a few iterations, before leveling out again. With different data sets (not displayed here), we tended to see the GRASP improve solutions a bit more steadily.

The right graph in figure 5 plots the solution of our BRKGA algorithm against the current iteration. We see that the BRKGA improves the solution quite slowly, eventually ceasing to improve around 120 iterations.We ran many more iterations for the BRKGA than the GRASP due to the significantly higher running times of the GRASP algorithm, as we show in the next section.

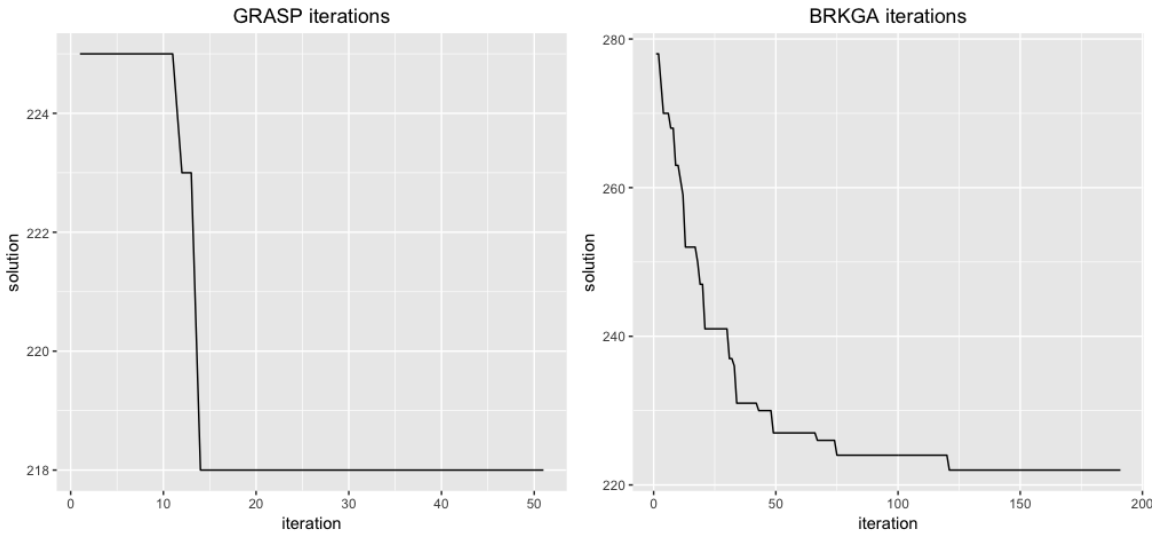

Figure 5: Best solution over iterations

6.3 BRKGA and GRASP running-time comparison

The left graph in figure 6 plots the solution of our GRASP algorithm against the running time. We see that after around 2,000 seconds the solution stops improving. The graph on the right plots the solution of our BRKGA algorithm against the running time. After roughly 420 seconds, the solution stops improving, however, after only 240 seconds, the solution no longer improves by much.

Comparing the two, we see that the BRKGA is much faster than the GRASP, although the GRASP finds a better solution.

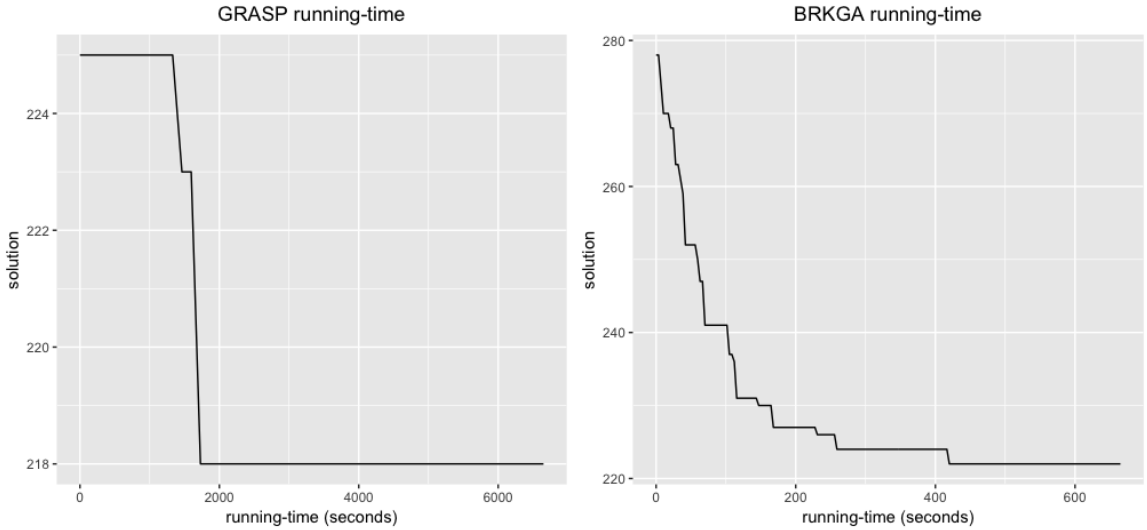

Figure 6: Best solution over time

7 CONCLUSION

First of all we can conclude that it was possible to model the given problem with all three approaches (ILP, GRASP, BRKGA). In general we expect that a change from one approach to another can be quite difficult as the ways of modeling the problem differ a lot.

In the problem statement we already made some assumptions, which turn out to be true. As expected the execution of an ILP can take a very long time, depending on the data. For the same data the meta-heuristics also find feasible and sufficiently good solutions in a shorter time. Therefore we claim that the use of meta-heuristics is often advisable (if the exact result is not necessary) because they usually scale much better for big datasets. Comparing the two meta-heuristics we could find out that the solutions of both meta-heuristics were similar. BRKGA found a sufficient solution in most cases in a shorter run time.

We could also find out that the results of the meta-heuristics rely heavily on their configuration. Therefore it is crucial to test out several parameters and optimize the algorithms before using them in a productive environment.

REFERENCES

[1] GONÇALVES, José Fernando; RESENDE, Mauricio GC. Biased random-key genetic algorithms for combinatorial optimization. Journal of Heuristics, 2011, 17. Jg., Nr. 5, S. 487-525.

[2] RESENDE, MAURICIO GC. Greedy randomized adaptive search procedures (GRASP). AT&T Labs Research Technical Report, 1998, 98. Jg., Nr. 1, S. 1-11.

Appendices

A

```
# Set 1
nNurses = 20
hours =  24
demand = [ 2,2,1,1,1,2,2,3,4,6,6,7,5,8,8,7,6,6,4,3,4,3,3,3 ]
minHours = 5
maxHours = 9
maxConsec = 3
maxPresence = 14

# Set 2
nNurses = 60
hours =  24
demand = [ 4,5,9,9,9,6,12,12,13,13,16,16,17,19,12,12,14,10,10,8,8,6,6,6 ]
minHours = 4
maxHours = 10
maxConsec = 4
maxPresence = 12

# Set 3
nNurses = 120
hours =  24
demand = [ 15,20,20,20,22,25,35,40,40,40,40,45,45,40,35,35,28,26,22,22,18,18,18,15 ]
minHours = 4
maxHours = 10
maxConsec = 5
maxPresence = 12

# Set 4
nNurses = 220
hours =  24
demand = [ 24,28,28,28,28,29,41,48,49,53,54,59,60,58,55,55,54,49,49,48,40,35,30,28 ]
minHours = 4
maxHours = 10
maxConsec = 5
maxPresence = 12
```

```
# Set 5
nNurses = 100
hours =  24
demand = [ 18,19,17,18,20,21,23,24,24,28,32,32,33,30,29,28,27,26,25,20,18,18,18,19 ]
minHours = 4
maxHours = 10
maxConsec = 5
maxPresence = 13

# Set 6
nNurses = 400
hours =  24
demand = [ 75,75,80,80,82,83,84,90,90,91,91,92,92,94,93,93,92,84,80,70,70,65,65,60 ]
minHours = 4
maxHours = 10
maxConsec = 5
maxPresence = 13
```