Course Project of Algorithmic Methods for Mathematical Models (AMMM)

Qiaorui Xiang
Fangyuan Zhao

Facultad de Informática de Barcelona

de Universitat Politècnica de Catalunya

1 January 2018

Contents

1 Problem Statement

In this project, we need to design a working schedule for a set of nurses in a way that the number of nurses required is minimized. The problem requires that the number of nurses working each hour must fulfill the demand for that hour. In addition, there are some limitations on how the working hours should be scheduled. The working hours of nurses must be within a range: between *minimum hour* and *maximum hour*. The nurses must take an one hour break when they reached *maximum consecutive hour*. Furthermore, the nurses can only stay in the hospital for *maximum presence hour*.

2 Formulation of Problem

Input data

- $numNurses$: total number of nurses
- $hours$: total hour
- $minHours$: minimum hour should a single nurse be working
- $maxHours$: maximum hour can a single nurse be working
- $maxConsec$: maximum consecutive hour can a single nurse be working
- $maxPresence$: maximum hour can a single nurse be in the hospital
- $demand_h$: number of nurses required for hour h

Decision variables

- $works_{n,h}$: nurse n is working in hour h
- $working_n$: nurse n works

Math formulation

$$\boxed{Min : \sum_{n \in N} working_n} \tag{1}$$

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

$$\sum_{h \in H} works_{n,h} \geq working_n * minHours \quad \forall n \in N \tag{3}$$

$$\sum_{h \in H} works_{n,h} \leq working_n * maxHours \quad \forall n \in N \tag{4}$$

$$\sum_{i \in [h, h+maxConsec]} works_{n,i} \leq maxConsec \quad \forall n \in N \quad \forall h \in [1, hours - maxConsec] \quad (5)$$

$$\sum_{i \in [h+maxPresence, hours]} works_{n,i} \leq (1 - works_{n,h}) * hours \quad \forall n \in N \quad \forall h \in [1, hours - maxPresence] \quad (6)$$

$$\sum_{i \in [h+3, hours]} works_{n,i} \leq (1 - works_{n,h} + works_{n,h+1} + works_{n,h+2}) * hours \quad \forall n \in N \quad \forall h \in [1, hours - 3] \quad (7)$$

Constraint description

1. The objective function is the sum of working nurses. Since $working_n$ is a binary decision variable and takes value 1 only if nurse n is working, the sum of $working_n$ would give the number of working nurses.

2. The number of nurses working in each hour must meets the demand of that hour. $works_{n,h}$ is a binary decision variable, and it is 1 when nurse n is working in hour h. Therefore, for each hour, we sum $works_{n,h}$ over every nurse, and the value has to be greater or equal to $demand_h$.

3. Nurses should work at least for a minimum amount of hours, $minHours$. For each nurse, we sum $works_n$ over every hour. This gives the total working hours of each nurse, which should be greater than the minimum working hours required.

4. Nurses should work at most for maximum amount of hours which is $maxHours$, and the total working hours of each nurse should be less than that value. By multiplying $working_n$ with $maxHours$, we ensure that, if nurse n works, $working_n$ is 1.

5. Nurses should work consecutively for at most $maxConsec$ hours. To check this constraint, we count how many hours each nurse works in $maxConsec$+1 time range. The hours should be less than $maxConsec$.

6. Nurses can only stay in the hospital for certain amount of hours. If the nurse is not working at hour h, the equations always holds. However, once the nurse starts working, the right hand becomes 0. The left hand ensures the nurse does not work after hour h + $maxPresence$.

7. Nurses cannot rest for more than one consecutive hour. In this case, if nurse n works at hour h and rest at hour h+1 and hour h+2, the right hand side of the equation becomes 0, which forces nurse n to rest from hour h+3 to the end.

3 GRASP

Algorithm 1 GRASP

function LOCAL_SEARCH(C, sol, α)
  $sol' \leftarrow sol$
  $D' \leftarrow D$
  Evaluate efficiency $ef(a)\ \forall a \in sol$
  $\underline{s} \leftarrow min\ \{\ ef(a) \mid a \in sol\ \}$
  $\bar{s} \leftarrow max\ \{\ ef(a) \mid a \in sol\ \}$
  $sol = sol \setminus \{\ s \in sol \mid ef(a) \leq \underline{s} + \alpha(\bar{s} - \underline{s})\ \}$
  UPDATE(D')
  $neighbor \leftarrow$GREEDY_CONSTRUCT with D'
  $sol' = sol' \cup neighbor$
  REMOVE_SURPLUS_ASSIGNMENT(sol')
  UPDATE_BEST_SOLUTION((sol, sol'))
end function

function GREEDY_RANDOMIZED_CONSTRUCT(C, α, $seed$)
  $sol \leftarrow \emptyset$
  while $D \neq \emptyset$ **do**
    Evaluate greedy cost $gc(c)\ \forall c \in C$
    $\underline{s} \leftarrow min\ \{\ gc(c) \mid c \in C\ \}$
    $\bar{s} \leftarrow max\ \{\ gc(c) \mid c \in C\ \}$
    $RCL \leftarrow \{\ s \in C \mid gc(c) \geq \bar{s} - \alpha(\bar{s} - \underline{s})\ \}$
    select s randomly from RCL
    $sol = sol \cup s$
    UPDATE(D)
  end while
  return sol
end function

function GRASP($maxIteration$, α, $seed$)
  $opt \leftarrow \infty$
  Initialize the candidates set: C ▷ All possible working schedule
  while $maxIteration$ is not reached **do**
    $sol \leftarrow$ GREEDY_RANDOMIZED_CONSTRUCT(C, α, $seed$)
    LOCAL_SEARCH(C, sol, α)
    UPDATE_BEST_SOLUTION((sol, opt))
  end while
  return opt
end function

Candidate set C

Restricted candidate list is selected from candidate set C, which is a set of all non-constraint-violated working schedules for nurses. This is achieved by generating a binary tree, cutting any branches that violates any constraint. The depth of the tree is $min\ \{maxHour * 2 - 1,\ maxPresence\}$. In this way, solutions given by greedy procedure are always feasible.

Greedy function

Our greedy function is:

$$gc(c) = \sum_{h \in H} demand_h * c_h$$

Since candidate $c \in C$ is always a valid assignment, and C contains all the possible assignment, we get how many demands each assignment can fulfill by multiply each c_h with the demand. Then choose the one that fills the most at each stage.

RCL equation

As we define greedy cost as amount of demand fulfilled, we are finding assignment with the highest value. Thus the RCL equation seems to be reversed comparing to the standard version. $RCL \leftarrow \{\ s \in C \mid gc(c) \geq \bar{s} - \alpha(\bar{s} - \underline{s})\ \}$

Efficiency function

In the local search, we evaluate each assignment for its efficiency. The efficiency function is:

$$ef(a) = \sum_{h \in H} (remaining\_demand_h == 0) * a_h$$

Same as the greedy function, higher value indicates assignment is more efficient. The value of the function indicates how many demands would be unfulfilled if an assignment was eliminated. The assignment with the lowest value is the least efficient and will be taken out. Please be aware that remaining demand D' can have negative if the number of working nurses are more required.

4 BRKGA

Algorithm 2 BRKGA

```
1: function DECODE_CHROMOSOME(chr, C)
2:     sol ← ∅
3:     for i ∈ [0, numNurses − 1] do
4:         candidate_code ← chr[i]
5:         pivot_code ← chr[i + numNurses]
6:         a ← PARSE(C, candidate_code, pivot_code)          ▷ assignment
7:         sol = sol ∪ a
8:     end for
9:     UPDATE(D)
10:    rd ← REMAINING_DEMAND(D)
11:    if rd > 0 then
12:        fitness ← numNurses + rd
13:    else
14:        REMOVE_SURPLUS_ASSIGNMENT(sol)          ▷ simple local search
15:        fitness ← COST(sol)
16:    end if
17:    return sol, fitness
18: end function

19: function DECODE(P, C)
20:    for all chr ∈ P do
21:        solution, fitness ← DECODE_CHROMOSOME(chr, C)
22:        chr ← UPDATE(chr, solution, fitness)
23:    end for
24:    return P
25: end function
```

Chromosome structure

Each chromosome takes the size of $numNurses * 2$. The first half of the genes determines the index of candidate $c \in C$ while the last half of the genes are used to encode which hour should this c starts working.

To determine which candidate to choose, we calculate the segment of each candidate by computing $1/numCandidate$. Each candidate $c \in C$ is indexed. Then using $gene/segment$, we get an integer which is the index of the candidate we will choose.

To determine which hour to assign the candidate, we use $1/hours$ to get the segment of each hour. After, with $gene/segment - len(c)/2$ we get a hour $h \in H$ and left shifted half of length of c. With this we can get a *pivot* point which is where we want to insert the chosen candidate.

Finally, with combination of candidate c and pivot p, we can generate a complete assignment a.

5 Benchmark Results

Instance description

The problem size is defined as the amount of $hours$, and other parameters vary based on the change in problem size. The instances are generated in the size range of 8 to 99 $hours$, with 10 random instances for each size. Therefore, we get a benchmark of 920 instances. This is to guarantee that our results reflects the average performance.

Result

All computations are carried out on a personal computer Intel Core i7-4770 with 3.4 GHz processor and 8.00 GB RAM. The parameter used for each algorithm are shown in Table 1 below.

	Timeout (s)	**Parameter**
ILP	1800	
GRASP	600	$maxIteration = 50$, $\alpha = 0.2$
BRKGA	600	$maxGen = 100$, $p = 100$, $p_e = 0.15$, $p_m = 0.2$, $p_i = 0.7$

Table 1: Algorithm Parameters

First, we compare the performance of all ILP, GRASP, and BRKGA. Eventually, when the problem size gets large enough, IPL will take forever to solve it. Thus, we set a timeout time at 30 minutes. During our benchmark test, ILP encounters timeout very often after the problem size reaches 46. Therefore, it is only reasonable to compare the performance from 8 to 46 $hours$.

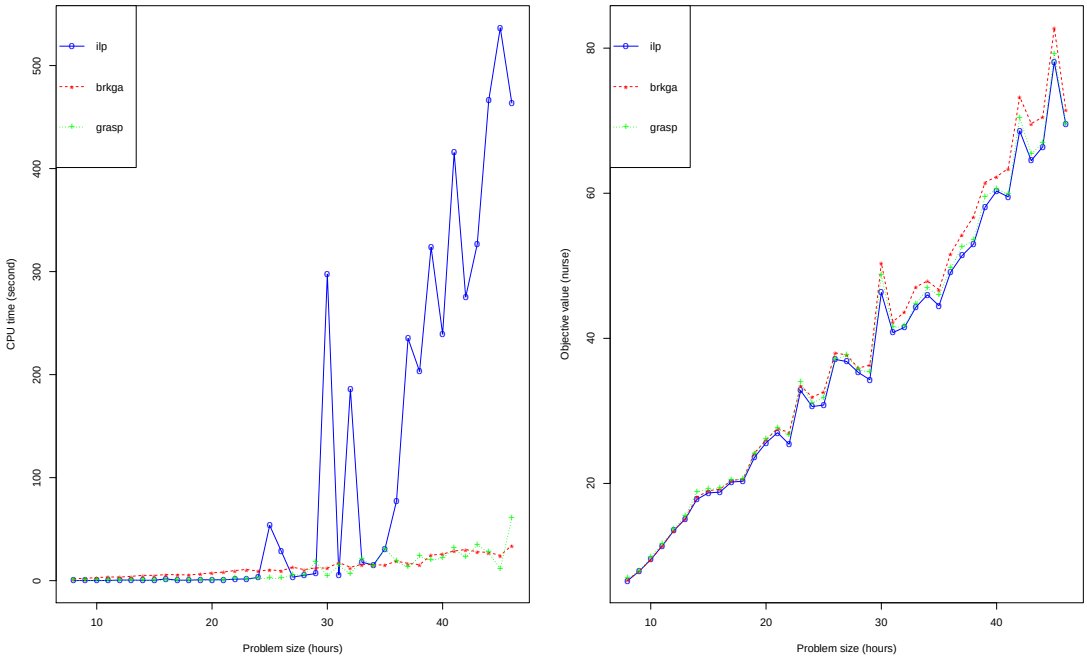

Figure 1: ILP vs BRKGA vs GRASP

As you can see in Figure 1, after the problem size reaches 25, the CPU time of ILP becomes unstable, and is increasing rapidly as the problem size becomes larger. However, the two Meta-heuristics are stable and fast. For the solution, the ILP always gives the optimal value. Although the solutions BRKGA produced are not that optimal, GRASP's solutions are generally close enough to the optimal. Along with the time consumption, GRASP provides a close-to-optimal solution but saving much more time.

Then, for instances with larger problem sizes, we compare the performance of the two meta-heuristics.

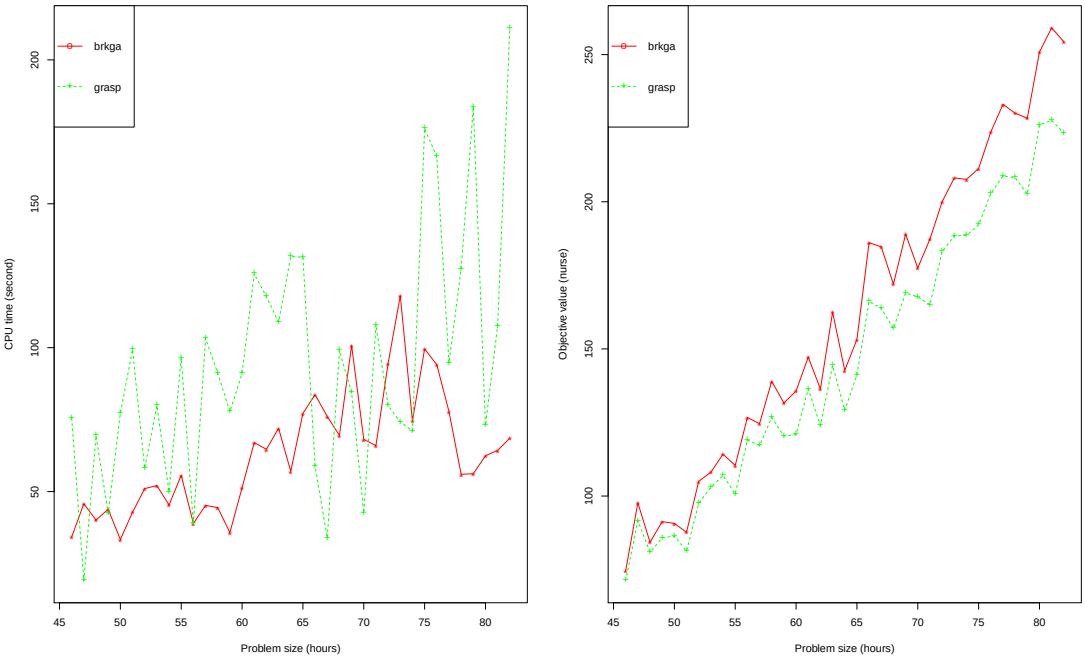

Figure 2: BRKGA vs GRASP

As shown in Figure 2, we can conclude that the CPU time of both GRASP and BRKGA are increasing overall, but GRASP is very unstable. On the other hand, BRKGA is more stable and consumes less time on average. In terms of the quality of the solutions, GRASP does produce more optimal solutions, especially towards large size problems. We can see in the figure that the difference gets large after 65.

The result only reflect this particular case scenario. The two heuristics may perform differently with other kind of problems.