Course project

Shams Methnani & Rodrigo Arias Mallo

January 10, 2018

1 Problem statement

Our problem to solve is to optimally assign nurses to working hours in a hospital such that demand for each hour is met and the fewest number of nurses are used.

This is an NP-hard problem in combinatorial optimization. It can be formulated as an integer program.

We are given a set of nurses, a specification of the demands for each hour as well as a set of constraints that must be satisfied regarding how long each nurse can work.

1.1 Parameters

The problem is described by the following set of parameters:

N	The number of nurses available
H	The hours in a day
$demand_h$	Demand for each hour h
$maxHours$	Max. number of hours each nurse can work
$minHours$	Min. number of hours each nurse can work
$maxConsec$	Max. consecutive hours each nurse can work
$maxPresence$	Max. hours present at the hospital

1.2 Decision variables

For the problem, we used three binary matrices P, W and S of size $N \times H$ as decision variables. The last one is auxiliary. The matrix P has the element $P_{n,h} = 1$ if the nurse n is present at the hospital at hour n, also if is working, $W_{n,h} = 1$, otherwise 0. The matrix S is an auxiliary matrix, with the element $S_{n,h} = 1$ if the nurse n starts at the hospital at the hour h, the first hour that is present, otherwise 0.

1.3 Formal definition

Minimize the objective function:

$$\sum_{n=1}^{N}\sum_{h=1}^{H} S_{n,h}$$

Subject to the following constraints:

Constraint 1 At least $demand_h$ nurses should be working at the hour h.

$$1 \leq h \leq H, \quad \sum_{n=1}^{N} W_{n,h} \geq demand_h$$

Constraint 2 Each working nurse should work at least $minHours$.

$$1 \leq n \leq N, \quad \sum_{h=1}^{H} W_{n,h} \geq minHours \cdot \sum_{j=1}^{H} S_{n,j}$$

Constraint 3 Each nurse should work at most $maxHours$.

$$1 \leq n \leq N, \quad \sum_{h=1}^{H} W_{n,h} \leq maxHours$$

Constraint 4 Each nurse should work at most $maxConsec$ consecutive hours.

$$1 \leq n \leq N, \quad 1 \leq h \leq H - maxConsec, \quad 1 \leq k \leq maxConsec,$$

$$\sum_{k} W_{n,h+k} \leq maxConsec$$

Constraint 5 No nurse can stay at the hospital for more than $maxPresence$ hours.

$$1 \leq n \leq N, \quad \sum_{h=1}^{H} P_{n,h} \leq maxPresence$$

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

$$1 \leq n \leq N, \quad 1 \leq h \leq H - 1,$$

$$P_{n,h} - W_{n,h} + P_{n,h+1} - W_{n,h+1} \leq 1$$

Constraint 7 Working nurses can start at most once.

$$1 \leq n \leq N, \quad \sum_{h=1}^{H} S_{n,h} \leq 1$$

Constraint 8a If a nurse is present at hour h and wasn't at previous hour, then started at current hour h.

$$1 \leq n \leq N, \quad 2 \leq h \leq H,$$

$$S_{n,h} \geq 1 - P_{n,h-1} + P_{n,h} - 1$$

Constraint 8b Add case for the first hour, if is present, has started at the first hour.

$$1 \leq n \leq N,$$

$$S_{n,1} = P_{n,1}$$

Constraint 9 If a nurse starts at current hour, then was not present at previous hour.

$$1 \leq n \leq N, \quad 2 \leq h \leq H,$$

$$S_{n,h} \leq 1 - P_{n,h-1}$$

Constraint 10 If a nurse starts at current hour, then should be present at current hour.

$$1 \leq n \leq N, \quad 2 \leq h \leq H,$$

$$S_{n,h} \leq P_{n,h}$$

2 GRASP

The GRASP method is a meta-heuristic in which each iteration consists of two phases: construction and local search.

First, we need to specify the problem in terms of parts that can be selected. The ground set E contains elements that can be selected in order to build a solution. Let S be a possible solution by selecting parts of E, so $S \in 2^E$. We can build F as the set of solutions that are feasible $F \subseteq 2^E$

In our problem, we need to select the behavior of N nurses. A possible solution can be defined by the selection of elements that describe what a nurse n is doing at the time h. A tuple $t = (n, h, s)$ can code that the nurse

n should be at state s in the time h. The state can be 0 if the nurse is not at the hospital, 1 if is working, or 2 if is resting. A feasible solution S should consist of a selection of tuples t such that, they define the behavior of every nurse at every time, and also satisfy the previous constraints.

Furthermore, in order to build a solution, a notion of cost is needed to select among all the possible tuples that can be selected. Lets ignore the constraints in the cost function, and check the feasibility at the end. Iteratively we can select a naive solution by a simple procedure.

2.1 Constructive phase

We begin with an empty solution. Then we compute the cost of all the possible elements that can be added to the solution. A good candidate will try to meet the demand by adding nurses that are in the working state.

We create the restricted candidate list (RCL) by selecting working nurses with high probability. Then, one of the possible combinations will be chosen, optimistically assigning one nurse to work in that time. Then we continue adding nurses in a working state. Once the demand is met, then we now will prefer adding to the solution tuples that contain nurses in a non-working state. The probability of such state should now be added with high probability.

A small snippet of a simplified python pseudocode can be found below, describing the constructive phase as the `greedy_build` function.

```python
def greedy_build(self, alpha):
  solution = set()
  C = self.initial_candidates()
  while len(C) != 0:
    costs = self.evaluate_costs(C, solution)
    cmin = np.min(costs)
    cmax = np.max(costs)
    cmed = cmin + alpha * (cmax - cmin)
    RCL_indices = np.where(costs <= cmed)[0]
    si = np.random.choice(RCL_indices)
    s = C[si]

    self.select_candidate(solution, s)
    C = self.update_candidates(C, s)
  return solution
```

We see the algorithm follows closely the definition of the GRASP meta-heuristic. The RCL equation, is almost the same.

$$RCL = \{e \in C \mid c(e) \leq c_{min} + \alpha(c_{max} - c_{min})\}$$

The selection of the cost function is the core of the GRASP algorithm, so we need to take some care to guarantee a good behavior.

2.2 Cost function

The function was designed to select at each step the most promising element. Is based on a set of rules to assign large weights to infeasible actions. All the elements in the set of candidates C are evaluated. A simplified version is shown below.

```
def evaluate_costs(self, C, solution):
  ...
  costs = np.zeros([len(C)], dtype='int')
  infos = []
  inf = 1000

  for i in range(len(C)):
    e = C[i]
    cost = 0
    info = ''
    n, h, s = e

    already_assigned = (n,h) in self.assigned

    working_time = self.worktime[n]

    present_hours = self.presencetime[n]

    work_done = self.workdone[h]
    demand_left = p.demand[h] - work_done
    hours_left = p.minHours - work_done

    can_work_consec = self.consec_table[n, h]

    already_started = present_hours >= 1
    is_next_hour_present = ((n,h+1,1) in solution
      or (n,h+1,2) in solution)
    is_previous_hour_present = ((n,h-1,1) in solution
      or (n,h-1,2) in solution)

    is_next_hour_home = (n,h+1,0) in solution
    is_previous_hour_home = (n,h-1,0) in solution
```

```
break_previous_hour = (n, h-1, 2) in solution
break_next_hour = (n, h+1, 2) in solution

# If we already have a assigned state, avoid
if already_assigned:
  costs[i] = inf
  continue

if s in {1, 2}: # Present at the hospital

  # If we are already the maximum time at the hospital, avoid
  if present_hours >= p.maxPresence:
    costs[i] = inf
    continue

  if already_started:

    # If already started, and is next and previous hour home,
    # avoid going work again

    # If is adding more consecutive hours, prefer
    if is_next_hour_present or is_previous_hour_present:
      # But only if is some demand at that point
      if demand_left > 0:
        cost -= 333

    # Otherwise, try to avoid
    else:
      costs[i] = inf + 4
      continue

if s == 0: # Home
  if already_started:

    # Delay going home if is not at the hospital
    if not is_next_hour_present:
      cost += 401

    else:
      cost += 401

  else:
    # Delay if didnt started
```

```
        cost += 500

    elif s == 1: # Working

      # Avoid more work than allowed
      if working_time >= p.maxHours:
        costs[i] = inf
        continue

      # Avoid working more than maxConsec hours
      if can_work_consec == 0:
        costs[i] = inf + 2
        continue

      # We want a new nurse proportional to the demand
      if demand_left > 0:
        cost -= demand_left * 100

      # Also try to reach minHours only if already started
      if hours_left > 0 and already_started:
        cost -= hours_left * 51

      # Try to avoid work if there is no demand
      if not already_started and demand_left <= 0:
        cost += 700

    elif s == 2: # Break
      # If was already resting, avoid at all costs
      if break_previous_hour or break_next_hour:
        costs[i] = inf
        continue

      # Only rest if it's better than work
      if already_started:
        cost += 50
      else:
        cost += 700

    costs[i] = cost

  return costs
```

2.3 Local search

Local search was not used, the greedy construction phase has been optimized to find a good solution by using an exhaustive search of possibilities. The pseudocode can be found below.

```
def local_search(self, solution):
  while not is_locally_optimal(solution):
    neighbors, costs = find_neigbors(solution)
    index_best = np.argmin(costs)
    solution = neighbors[index_best]
  return solution
```

2.4 Comments

The implementation of the problem in GRASP can be greatly improved by using a representation of the problem similar to the one used in BRGKA.

3 BRKGA

Genetic algorithms borrow mechanisms from nature, namely the concept of survival of the fittest, in order to search a large solution space. A population of solutions is evolved over several generations by iteratively breeding new solutions by mating or *crossing* individuals from previous generations. Biased Randomized-Key Genetic Algorithm (BRKGA) is such an algorithm in which certain members of the population are selected for crossover with higher probability.

We represent our problem with a set of individuals with an associated *chromosome*, each of which encodes a solution.

Each individual or chromosome has associated with it a fitness score, which will determine if they are an *elite individual* or *non-elite individual*. We can now add a bias to the selection of the parentage for next generation individuals the selecting a parent from the elite set with a higher probability in order to pass on desirable characteristics.

We used the parameters $p(elite) = 0.1$, $p(mutant) = 0.2$, $p(inherit) = 0.7$ and population number=10. The absolute maximum number of iterations is set to 10000, but if there is no improvement in the best fit, for the last 500 iterations, an early break happens.

3.1 Chromosome structure

The solution is codified by using three scalars p, s and b and a vector v of size H for each nurse. The scalars codify the number of hours p that the nurse is present at the hospital, the first hour s at which is at the hospital, and the number of breaks b while is present.

The vector v is used from 1 to the number of present hours p. Each element stores a score that will be used to determine the preferred hours in which a break should be set. Lower scores will be selected first, filling the breaks up to b.

The decision whether a nurse works or not, is based on p, if the presence hours are less than $minHours$ the nurse doesn't start working.

The chromosome structure is then matrix of N rows and $3 + H$ columns, but flattened into a long vector of size $N \times 3 + H$.

3.2 Decoder

The decoder part of the algorithm, which is the only part that depends on the problem, computes the fitness of each individual in the population. The fitness $f(p, s, b, v)$ is a heuristic value that estimates the objective, and has to be minimized. Is computed by adding five badness values BN, BD, BH, BC and BF together.

In order to compute the values, we first need to determine when each nurse works. By using p and s we have the interval that they are present. And using the indices of the first b lower values of v, we now where they rest.

The number of nurses needed is the first value BN, which takes into account that we want to minimize the number of nurses used.

The value BD measures the demand left that is not met. Is computed at each hour h as the difference of the problem demand and the nurses offer (the working hours provided by the working nurses) as $BD = 100 \sum_h \text{demand}_h - \text{offer}_h$.

The value $BH(n)$ measures the error of each nurse n, when they work more than $maxHours$ or when they work less than $minHours$, we use $w = p - b$ to compute the number of working hours. If they work more hours than allowed, $BH(n)$ is set to the number of exceeding hours $w - maxHours$. If they work less than allowed, $BH(n)$ is set to the number of hours left $minHours - w$. Finally, we get $BH = \sum_n 200 + 20BH(n)$.

Note that we use the floating point representation of the values p, s, b and v when they are part of the fitness function. So we can allow that lower differences of each gene produce a change in the fitness function. Otherwise, we use the integer representation of the parameters.

The value BC test if there is any problem with the maximum number of consecutive hours that each nurse is working. For each nurse n the maximum number of consecutive hours M is computed, and if is greater than $maxConsec$, $BC(n)$ is set to $100(M - maxConsec) - 10b$. We use the breaks to prefer those solutions that include more breaks. Finally we have $BC = \sum_n BC(n)$.

The value BF measures the distance of the non-working nurses to the hours that have some demand left. The distance is computed for each hour with demand left h to the closest hour c that the nurse would start working if p was at least $minHours$. Is the start hour s when $s > h$ and the end hour $s+p$ when $s+p < h$. So, if the nurse don't cover the hour h, $BF(n) = h - c$. However, if the nurse could cover the hour, without changing start, we only need to take into account the distance to minHours, so $BF(n) = minHours - p$. The final value is the sum for all nurses that are non-working, $BF = \sum_n BF(n)$

The final fitness value is just the sum of all the intermediate values:

$$f(p, s, b, v) = BN + BD + BH + BC + BF$$

The fitness is stored in an array, computed for each individual of the population, and returned to the BRKGA algorithm, in order to continue the iteration.

4 Results

The three solvers were compared solving equal problems, with a big number of nurses. The CPLEX solver was limit to 1 minute of execution time, and the best solution found was used. In the table 1 we can see the execution time in seconds, as well as the best objective value found, the number of nurses needed, for each method. A $*$ symbol is used to represent that the method did not found any solution. By setting the seed to the indicated value, a reproducible run should produce the same results (time may vary).

We see that the ILP method gets a good solution, even with larger instances. It is not able to compute the optimal solution, but the one found has a very low objective value. The GRASP method however, not always is able to find a solution and usually takes longer. The BRKGA method computes values close to the ILP method, in about the same time for 90 nurses. However, is not alway able to find one feasible.

5 Structure of the project

5.1 The OPL model

The ILP problem was first written in OLP by using a small example, and it can be found in the `opl/` folder. The `proj.mod` file contains the model of the problem, and the example data is stored in `proj.dat`. A small python script and a CPLEX batch file help the extraction of the solution. The Makefile is designed to use `oplrun` and `cplex` to solve the problem and present the solution. The `oplrun` program evaluates the model and builds the file `sol.lp`, which is solved by the `cplex` program. The script `cplex-save.txt` provide CPLEX the instructions to save the solution in `sol.xml` in order to be read by the python script `cplex-save.txt`:

```
% cd opl/
% make
...
cplex -f cplex-save.txt

Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.6.0.0
...
Root relaxation solution time = 0.00 sec. (0.15 ticks)
...
python read-sol.py
NStart
[[0 0 0 0 1 0]
 [1 0 0 0 0 0]
 [1 0 0 0 0 0]]
NWorking
[[0 0 0 0 1 1]
 [1 1 0 0 0 0]
 [1 1 0 0 0 0]]
NPresent
[[0 0 0 0 1 1]
 [1 1 0 0 0 0]
 [1 1 0 0 0 0]]
```

As we need to run and evaluate a lot of problems, this method is a bit complex, as requires the problem to be defined in the OPL data language. We opted for the use of the PuLP python package to communicate with CPLEX, in order to write the problem and read the solution as well as the objective value, more easily.

5.2 The PuLP model

The PuLP package provides a almost transparent interface to work with multiple solvers. We used the CPLEX solver by default, as it seems to be the fastest.

The same constraints in `opl/proj.mod` were translated into python. The file `src/lp.py` contains the ILP model in python. A class `Solution` provides a generic method `solve` to be implemented, as well as an instance to a problem to be solved.

5.3 Instance generator

The instance generator is built in the `Problem` class, located in `nurses.py`

Random instances of the problem are generated by using the parameters: `seed` the random seed, `N` the number of nurses and `H` the hours in a day, set by default to 24.

We used a random distribution, with more expected value in the daily hours, behaving similarly to a real case. Once a instance problem generated can be solved by any solver. An example problem:

```
% python nurses.py
Problem(seed=1, N=10, H=24)
-------------------------------------------------------
hour   0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
-------------------------------------------------------
dem.   1 1 0 0 1 0 0 2 1 2 2 2 3 2 3 1 1 0 1 0 0 0 1 0
-------------------------------------------------------
 maxConsec    : 7
 maxPresence : 9
 maxHours     : 8
 minHours     : 8
-------------------------------------------------------
```

6 How to run

The ILP, GRASP and BRKGA solvers are coded in a extension of the `Solution` class, placed in `src/`, in the files `lp.py`, `grasp.py` and `brkga.py`, respectively.

Each solver is iteratively chosen to solve the same problem. Once all the solutions are computed, a new random problem is generated and solved again. The file `main.py` produces the table 1. Can be executed as:

```
% python main.py
```

In order to execute the same results, fix the seed to the indicated in the table. A total of 10 runs is set by default, automatically incrementing the seed, and printing the intermediate table at each row.

		ILP		GRASP		BRKGA	
Seed	Nurses	Obj.	Time	Obj.	Time	Obj.	Time
8	90	41	61.1	52	62.9	*	83.1
9	90	39	61.2	*	619.3	42	71.2
10	90	39	61.2	56	62.2	42	51.6
11	90	31	61.2	43	63.4	44	46.2
12	90	39	61.2	46	127.4	41	96.5
13	90	36	61.1	*	638.2	*	81.3
14	90	39	61.1	51	95.3	*	71.1
15	90	31	61.2	42	32.2	43	65.3
16	90	35	61.1	56	189.3	44	45.5
17	90	35	61.2	57	31.5	42	63.4
18	90	37	61.8	*	378.5	45	54.0
19	90	31	61.1	46	31.9	33	28.3
20	90	31	61.1	43	15.0	38	19.6
21	90	33	61.1	*	324.4	*	54.9
22	90	40	61.1	*	479.2	*	119.4
23	90	*	1.5	*	472.3	*	76.2
24	90	32	61.1	46	17.5	*	26.7
25	90	31	61.1	46	17.2	*	37.0
26	90	40	61.1	60	175.3	39	67.1
27	90	35	61.1	*	353.6	37	48.7
30	150	60	61.9			*	236.9
31	150	60	61.9			68	140.0
32	150	56	61.9			64	200.1
33	150	66	61.9			*	186.0
34	150	67	61.9			72	293.6
35	150	53	61.9			73	161.5
36	150	75	61.9			77	152.3
37	150	58	61.8			63	216.3
38	150	62	61.8			64	219.8

Table 1: Results with the different methods. Time in seconds.