Report AMMM
Optimization Project

Ana Puertas, Jordi Piqué

AMMM

FIB • UPC

January 6, 2018

Abstract

In this paper we will solve an optimization combinatorial problem. This problem consist of assigning the hours a set of nurses will work in a hospital. They have to cover all the demand under certain constraints.

To solve that problem we will try 3 different methods. The first is modelling it creating a linear model and solving it using OPL, an linear programming solver from IBM. The last two are meta-heuristics. The first one is GRASP, a method that combines a random constructive algorithm with local search. The second one is BRKGA, a genetic algorithm.

The goal is to compare the goodness of the solution produced by each algorithm and their execution time.

Contents

1 Description of the problem

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.
- No nurse can rest for more than one consecutive hour.

The goal of this project is to determine at which hours each nurse should be working in order to **minimize the number of nurses required** and satisfy all the aforementioned constraints.

2 The Integer Lineal Model

2.1 First version

2.1.1 Variables

- $W_n(boolean)$ The nurse n works
- $WH_{n,h}(boolean)$ The nurse n works at hour h
- $S_{n,h}(boolean)$ The nurse n has started her working day before or at hour h
- $E_{n,h}(boolean)$ The nurse n has not ended her working day before hour h
- $DJ_{n,h}(boolean)$ The hour h of nurse n is within her working day
- $comienzo_n(Z^+)$ The hour the nurse n starts working
- $final_n(Z^+)$ The hour the nurse n stops working

In order to understand better what these variables mean and how they are related, lets see an example for $nurse_0$:

$WH_{0,h}$	000010110111011000000000
W_0	1
$S_{0,h}$	000011111111111111111111
$E_{0,h}$	111111111111111100000000
$DJ_{0,h}$	000011111111111100000000
$comienzo_0$	4
$final_0$	15

2.1.2 Objective function

As we can see, this objective function tries to minimize the number of nurses that work at the hospital.

$$MIN\left[\sum_{n=0}^{nNurses} W_n\right]$$

2.1.3 Constraints

1. **The number of nurses working at each hour must be greater or equal to the demand.**

$$\sum_{n=0}^{nNurses} WH_{n,h} \geq demand_h \quad \forall h \in [0, hoursDay)$$

2. **Each nurse should work at least $minHours$ hours.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \geq minHours * W_n \quad \forall n \in [0, nNurses)$$

3. **Each nurse should work at most $maxHours$ hours.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \leq maxHours \quad \forall n \in [0, nNurses)$$

4. **Each nurse should work at most $maxConsec$ consecutive hours.** In that case we impose that no segment of $maxConsec + 1$ hours of all nurses is full of working hours.

$$\sum_{h=0}^{maxConsec+1} WH_{n,start+h} \leq maxConsec \quad \forall start \in [0, hoursDay - maxConsec) \quad \forall n \in [0, nNurses)$$

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

$$final_n - comienzo_n \leq maxPresence \quad \forall n \in [0, nNurses)$$

6. **No nurse can rest for more than one consecutive hour.** In that case we forbid to have two consecutive resting hours during the working day of a nurse.

$$DJ_{n,h} + DJ_{n,h+1} \leq WH_{n,h} + WH_{n,h+1} + 1 \quad \forall h \in [0, hoursDay - 1) \quad \forall n \in [0, nNurses)$$

7. **Relation between W_n and $WH_{n,h}$.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \geq W_n \quad \forall n \in [0, nNurses)$$

$$\sum_{h=0}^{hoursDay} WH_{n,h} \leq W_n * hoursDay \quad \forall n \in [0, nNurses)$$

8. **Relation between $S_{n,h}$ and $WH_{n,h}$.**

$$S_{n,1} = WH_{n,1} \quad \forall n \in [0, nNurses)$$

$$S_{n,h-1} + WH_{n_h} \geq S_{n,h} \quad \forall h \in [1, hoursDay) \quad \forall n \in [0, nNurses)$$

$$S_{n,h-1} + WH_{n_h} \leq 2 * S_{n,h} \quad \forall h \in [1, hoursDay) \quad \forall n \in [0, nNurses)$$

9. **Relation between $E_{n,h}$ and $WH_{n,h}$.**

$$E_{n,hoursDay} = WH_{n,hoursDay} \quad \forall n \in [0, nNurses)$$

$$E_{n,h+1} + WH_{n_h} \geq E_{n,h} \quad \forall h \in [0, hoursDay - 1) \quad \forall n \in [0, nNurses)$$

$$E_{n,h+1} + WH_{n_h} \leq 2 * E_{n,h} \quad \forall h \in [0, hoursDay - 1) \quad \forall n \in [0, nNurses)$$

10. **Relation between $DJ_{n,h}$ and $E_{n,h}$ and $S_{n,h}$.** We calculate the intersection between $E_{n,h}$ and $S_{n,h}$.

$$E_{n,h+1} + S_{n,h} \geq 2 * DJ_{n,h} \quad \forall h \in [0, hoursDay) \quad \forall n \in [0, nNurses)$$

$$E_{n,h+1} + S_{n,h} \leq DJ_{n,h} + 1 \quad \forall h \in [0, hoursDay) \quad \forall n \in [0, nNurses)$$

11. **Relation between $comienzo_n$ and $S_{n,h}$.**

$$comienzo_n = hoursDay - \sum_{h=0}^{hoursDay} S_{h,n} \quad \forall n \in [0, nNurses)$$

12. **Relation between $final_n$ and $E_{n,h}$.**

$$final_n = \sum_{h=0}^{hoursDay} E_{h,n} \quad \forall n \in [0, nNurses)$$

2.2 Improvements

We can eliminate the variable $DJ_{n,h}$ if we rewrite the constraint that says that a nurse can't rest for more than one consecutive hour. In that case we forbid to find the pattern 100(0)*1(1 | 0)*.

$$WH_{n,h} \leq WH_{n,h+1} + WH_{n,h+2} + \frac{\sum_{k=h+3}^{hoursDay}(1 - WH_{n,k})}{hoursDay - h - 3} \quad \forall h \in [0, hoursDay-2) \quad \forall n \in [0, nNurses)$$

We can eliminate $S_{n,h}$, $E_{n,h}$, $comienzo_n$ and $final_n$ if we rewrite the constraint that says that a nurse can't stay at the hospital for more than $maxPresence$ hours. In that case we forbid to find a nurse that works in two different hours separated by a gap of $maxPresence$ or more hours.

$$WH_{n,h1} + WH_{n,h2} \leq 1 \quad \forall h1 \in [0, hoursDay - maxPresence)$$
$$\forall h2 \in [h1 + maxPresence, hoursDay) \quad \forall n \in [0, nNurses)$$

Reducing the number of variables used for the modelization of the problem **improves a lot the execution time** of the IBM solver. To solve the following instance of the problem, the first version of the linear model uses 30.591 variables and after more than 40 min it hasn't found the optimal solution. The second one uses 7.725 variables and takes 1 min 24 s to find the optimal solution (241 nurses).

```
nNurses=309;
minHours=3;
maxPresence=12;
maxHours=10;
maxConsec=6;
hoursDay=24;
demand=[105 95 91 105 102 93 102 99 102 105 107 98 103 102 96 103 100 95 109 102 100 99 93 99];
```

2.3 Final version

2.3.1 Variables

- $W_n(boolean)$ The nurse n works
- $WH_{n,h}(boolean)$ The nurse n works at hour h

2.3.2 Objective function

$$MIN\Bigg[\sum_{n=0}^{nNurses} W_n\Bigg]$$

2.3.3 Constraints

1. **The number of nurses working at each hour must be greater or equal to the demand.**

$$\sum_{n=0}^{nNurses} WH_{n,h} \geq demand_h \quad \forall h \in [0, hoursDay)$$

2. **Each nurse should work at least $minHours$ hours.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \geq minHours * W_n \quad \forall n \in [0, nNurses)$$

3. **Each nurse should work at most $maxHours$ hours.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \leq maxHours \quad \forall n \in [0, nNurses)$$

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

$$\sum_{h=0}^{maxConsec+1} WH_{n,start+h} \leq maxConsec \quad \forall start \in [0, hoursDay{-}maxConsec) \quad \forall n \in [0, nNurses)$$

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

$$WH_{n,h1} + WH_{n,h2} \leq 1 \quad \forall h1 \in [0, hoursDay - maxPresence)$$

$$\forall h2 \in [h1 + maxPresence, hoursDay) \quad \forall n \in [0, nNurses)$$

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

$$WH_{n,h} \leq WH_{n,h+1} + WH_{n,h+2} + \frac{\sum_{k=h+3}^{hoursDay}(1 - WH_{n,k})}{hoursDay - h - 2} \quad \forall h \in [0, hoursDay{-}2) \quad \forall n \in [0, nNurses)$$

7. **Relation between W_n and $WH_{n,h}$.**

$$\sum_{h=0}^{hoursDay} WH_{n,h} \geq W_n \quad \forall n \in [0, nNurses)$$

$$\sum_{h=0}^{hoursDay} WH_{n,h} \leq W_n * hoursDay \quad \forall n \in [0, nNurses)$$

3 The GRASP meta-heuristic

3.1 The constructive algorithm

The constructive algorithm has several approaches. One of them is building the set of nurses by **adding a complete nurse** every time and the other is building each nurse by **adding the hours she will work** and then adding this nurse to the set.

The second one has a problem. The constraints of $maxHours$, $restingHours$, $maxPresence$ and $maxConsec$ can be validated easily every time we add a new hour to the nurse. The problem is that we can't determine trivially if the $minHours$ constraint will be violated or not. If we try to add hours until the nurse has more than $minHours$ we can find a situation where the nurse works less than $minHours$ and we can't add more hours. Let's see an example.

$minHours = 6$
$maxHours = 8$
$maxPresence = 10$
$maxConsec = 2$

$WH_{0,h} = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

In that case we have the $nurse_0$ that stays at the hospital during 10 hours and works a total of 5 hours. We need to add one more hour to reach the minimum of 6 hours. The problem is that we can't add another hour to the beginning or end of the working day because $maxPresence$ is 10 and the nurse already stays at the hospital 10 hours. The nurse can't work in some of the hours where she rests because then there will be a group of 3 consecutive working hours and we have $maxConsec = 3$.

It's true that this kind of situation is not usual and we could try to build another time this nurse. However, there exists another solution.

This solution is the first approach described in the first pharagraph of this section. It consist of adding directly a feasible nurse to the set of nurses. For doing that, we need first to generate all the feasible nurses. This can be done easily by brute force. Because of the maximum size of the working day (only 24 hours) this problem, despite having an exponential complexity, can be managed.

We don't have to generate all the possible nurses. We only need to generate all the feasible nurses that start at the first hour. The other nurses can be created by shifting the nurses that start at the first hour. The working day of the nurse will have a maximum length of $maxPresence$.

```
procedure generatedFeasibleNursesStart0(minHours, maxHours, maxConsec, maxPresence) {
    setWorkingDays = {}
    for (nHours=minHours; nHours<=maxHours; ++nHours) {
        workingDays = recursiveGenerator([1], nHours-1)
        setWorkingDays.add(workingDays)
    }
}
```

```
procedure recursiveGenerator(workingDay, nHours, maxConsec, maxPresence) {
    if (not isOkMaxConsec(maxConsec, workingDay)) {
        return {}
    }

    else if (nHours == 0) {
        return {workingDay}
    }

    else if (workingDay.size() - 1 == maxPresence) {
        setWorkingDays = {}
        setWorkingDays.add(recursiveGenerator(workingDay + [1], nHours-1))
        return setWorkinDays
    }

    else if (workingDay.size() - 2 <= maxPresence) {
        setWorkingDays = {}
        setWorkingDays.add(recursiveGenerator(workingDay + [1], nHours-1))
        setWorkingDays.add(recursiveGenerator(workingDay + [0,1], nHours-1))
        return setWorkinDays
    }

    else {
        return {}
    }
}
```

For the constructive algorithm, we don't calculate the greedy cost for all the feasible nurses, because in some cases we can have hundreds of thousands of feasible nurses. Instead of that, we take a random sample of all the possible nurses and then we calculate their greedy cost. The advantage is the reduction of execution time. The only problem is that alfa can no longer control the randomness of the constructive algorithm, due to the randomness of the process of taking samples in the whole set of feasible nurses.

```
procedure construct(alfa, demand, minHours, maxHours, maxConsec, maxPresence, hoursDay) {
    feasibleNursesStart0 = generatedFeasibleNursesStart0
                                (minHours, maxHours, maxConsec, maxPresence, hoursDay)
    setNurses = emptySet
    while (sum(demand) > 0) {
        randomSelectedNurses = selectNursesStartingAnyHour(feasibleNursesStart0)
        listCosts = calculateGreedyCost(randomSelectedNurses)
        RCL = createRCL(listCosts, alfa)
        nurse = selectRandom(RCL)
        setNurses.add(nurse)
        demand = updateDemand(demand)
        }
    return setNurses
}
```

3.1.1 The greedy cost

The greedy cost of a nurse is the sum of the uncovered demand of the hours when she doesn't work. This cost tries to reward those nurses that work during the hours with more demand that is still uncovered.

$$greedyCostNurse = \sum_{h=0}^{hoursDay} (1 - workNurse_h) * uncoveredDemand_h$$

3.2 The local search algorithm

The local search algorithm tries to remove every nurse. When it removes a nurse it tries to redistribute, if necessary, the hours this nurse worked to the remaining ones. If the redistribution of hours can't be done then, the removed nurse is added another time.

```
procedure construct(demand, setNurses) {
    forall nurse in setNurses {
        setNurses.remove(nurse)
        success = setNurses.redistribute(nurse)
        if not success{
            setNurses.add(nurse)
        }
    }
    return setNurses
}
```

3.3 Optimizations

3.3.1 Parallel

GRASP is a meta-heuristic that allows the programmer to parallelize it easily. The total number of independent executions of GRASP can be divided among several threads. Each thread computes $totalNumberExecutions/numberThreads$ executions. Then the main thread takes the best result of all the threads and that's the final result.

4 The BRKGA meta-heuristic

4.1 The chromosome structure

The chromosome encodes the set of nurses. Each chunk of $maxLenEncNurse$ length encodes a nurse. The first element of that chunk says if the nurse works or not. The second element says at which hour the nurse starts working. If this element is greater than 0.5, then the starting hour and the nurse are reversed. The remaining elements represent the length of every chunk of consecutive working hours.

0.67 The nurse works (> 0.3)
0.80 The nurse starts her reversed working day, for example, at hour 20.
0.75 She works $0.75 * maxHours$ (if it's less than $maxConsec$) in the first chunk of consecutive hours.
0.25 She works $0.25 * maxHours$ (if it's less than $maxConsec$) in the second chunk of consecutive hours.

If we take $maxHours = 8$ and $maxConsec = 6$ then, the decoded nurse will be:

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0

4.2 The decoder algorithm

The decoder produces the set of nurses. Each nurse of this set doesn't violate any of the constraints. The only constraint that could be violated is that the offer of nurses must be greater or equal to the demand. In principle, it's impossible to produce a completely feasible set of nurses in polynomial time because if we could do that we'll be solving an NP problem.

4.2.1 The fitness

The positive difference between the demand and the offer ($demand_h - offer_h$) is used to calculate the fitness of the solution. The fitness is calculated as follows:

$$uncoveredDemand = \sum_{h=0}^{hoursDay} max(0, demand_h - offer_h)$$

$$extraOffer = \sum_{h=0}^{hoursDay} (offer_h - demand_h) \quad if \quad offer_h > demand_h$$

$$fitness = uncoveredDemand * nNurses^2 + extraOffer + nNursesThatWork$$

A low fitness is better than a higher one.

That force the BRKGA to reduce first the positive difference between the demand and the offer and, when all the demand is covered, to start reducing the number of nurses that work. The extraOffer helps to balance the extra number of nurses and this allows the algorithm to find feasible and better solutions faster.

4.2.2 Tricks

In order to find a good solution faster, there are some tricks used to interpret the chromosome.

The **first** one is being sure that almost all the constraints are not violated. That helps to discard a lot of unfeasible configurations of nurses.

The **second** one is increasing the probability that a nurse will work. Because the initial chromosome is a random set of real numbers between 0 and 1, if the number that says if a nurse work was interpreted in the following way:

$$number < 0.5 \rightarrow \neg work$$

$$number \geq 0.5 \rightarrow work$$

half of the nurses will not work at the first iteration and that gives a lower chance to cover all the demand. That's why this threshold has been set to 0.3.

$$number < 0.3 \rightarrow \neg work$$

$$number \geq 0.3 \rightarrow work$$

The **third** one is interpreting the $initHour$ in a way such that the offer is balanced according to the demand. For example, if we have a constant demand and a nurse has the same probability of starting at any hour, then the central hours will have a big offer of nurses compared to the beginning of the day or the end. That's because at the first hour will work only the nurses starting at the first hour, but at the fifth hour will probably work most of the nurses starting from the first to the fifth hour. In that way, the fifth hour will have approximately 5 times the number of nurses that work in the first hour. The decoder tries to create different thresholds in order to balance the difference between the offer of nurses and the demand of nurses for all hours. For example, in a constant demand, the thresholds could be the following:

1. $0 \leq x < 0.23$
2. $0.23 \leq x < 0.28$
3. $0.28 \leq x < 0.31$
4. ...

4.2.3 The pseudo-code

```
procedure decoder(population, params) {
    distrInitHour = getDistrInitHour(params)
    lengthEncodedNurse = getLengthEncodedNurse(params)
    foreach (individual in population) {
        setNurses = {}
        setEncodedNurses = divideIndividual(individual, lengthEncodedNurse)
        foreach (encodedNurse in setEncodedNurses) {
            works = encodedNurse[0] > 0.3
            initHour = getInitHour(distrInitHour, encodedNurse[1])
            listLengthConsecHours = getChuncksConsecHours(encodedNurse[2:], params)
            if (works) {
                nurse = generateNurse(initHour, listLengthConsecHours)
            }
            else {
                nurse = [0] * params.hoursDay
            }
            setNurses.add(nurse)
        }
        offer = getOffer(setNurses)
        uncoveredDemand = max(0, demand_i - offer_i) forall demand, offer
        extraOffer = (offer_i - demand_i)^2 forall demand, offer s.t. offer_i > demand_i + 1
        nWorkingNurses = getNursesWork(setNurses)
        fitness = uncoveredDemand*(params.nNurses)^2 + extraOffer + nWorkingNurses
    }
}
```

5 Results

For the executions, 50 instances of increasing difficulty have been created using the **feasible 1** generator, because it generates instances that are quite hard to solve but all of them are feasible. If a solver doesn't find a solution we know that the responsible is the solver, not the instance.

We have done 3 rounds of executions. In the first round the solvers have 5 min, in the second round 10 min and in the last round 20 min. The executions have been done in a laptop with an **Intel Core i7-6700HQ 2.60GHz** processor, **16GB** of memory and with **Microsoft Windows 10 Enterprise** as the OS. For more details about the results of the executions, you can look at the appendix.

5.1 OPL

The **OPL** solver has run with the default parameters except the memory, that has been increased to 12GB.

It only has advantage over the meta-heuristics for problems with a reduced size. The increase of computational time over the 3 runs gives a noticeable performance in the solutions for the first 20 instances. For the large instances, an increase in computational time doesn't affect much the cost of the solution.

5.2 GRASP

The **GRASP** has run with $numIter$ and $maxItWithoutImpr$ to infinite, $alpha = 0.1$ and 8 threads.

It is the solver that has a better performance when the ratio $\frac{sizeOfTheProblem}{computationalTime}$ is very high. With only 5 min of computational time, it can find a feasible solution for all the instances. It is extremelly good for finding quickly a feasible solution. The problem is that it doesn't improve a lot the solution if we give it more computational time.

5.3 BRKGA

The **BRKGA** has run with 200 individuals, $maxGenerations = maxItWithoutImpr = \infty$, $eliteProp = 0.1$, $mutantsProp = 0.1$ and $inheritanceProp = 0.7$.

It is between the GRASP and the OPL. It has some learning process and that's why it can continue to improve the solution if we give it more computational time. The main problem for BRKGA is that it only makes sense to use it starting from a medium size problem, because for small problem OPL is better, but for medium an large problems, GRASP is better than BRKGA. BRKGA is better than GRASP for small problem, but, as we have said before, for small problems OPL is the best.

However, BRKGA has a disatvantage over OPL and GRASP. The last are parallelized whereas

BRKGA run sequentially. The only part of BRKGA that can be parallelized is the decoder. The decoding of each individual is independent and can be done in parallel. The problem is that the implementation of the decoder has been written in Python and the parallel version of the decoder is slower than the sequential version. That's because Python creates a pool of processes in order to parallelize and they have a big overhead when the main process has to share data to the other processes. In the case of BRKGA, the increase of performance due to the parallelization doesn't compensate this overhead. If BRKGA could be parallelized without this overhead, it will probably win GRASP in most of the instances.

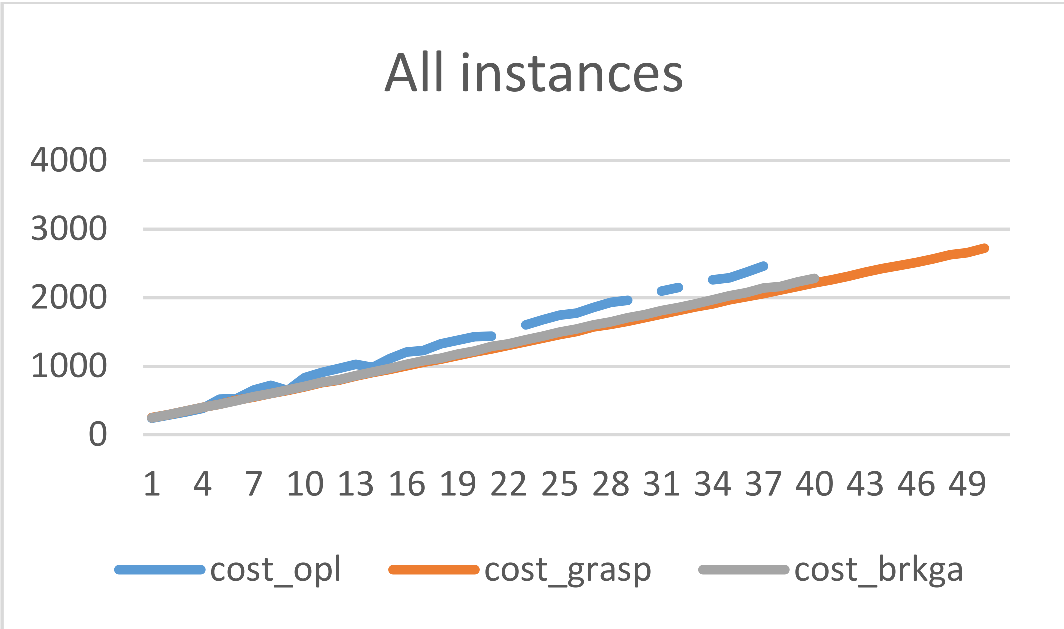


Figure 1: Cost of the solution found by the 3 solvers for all the 50 instances with maxTime = 5 min

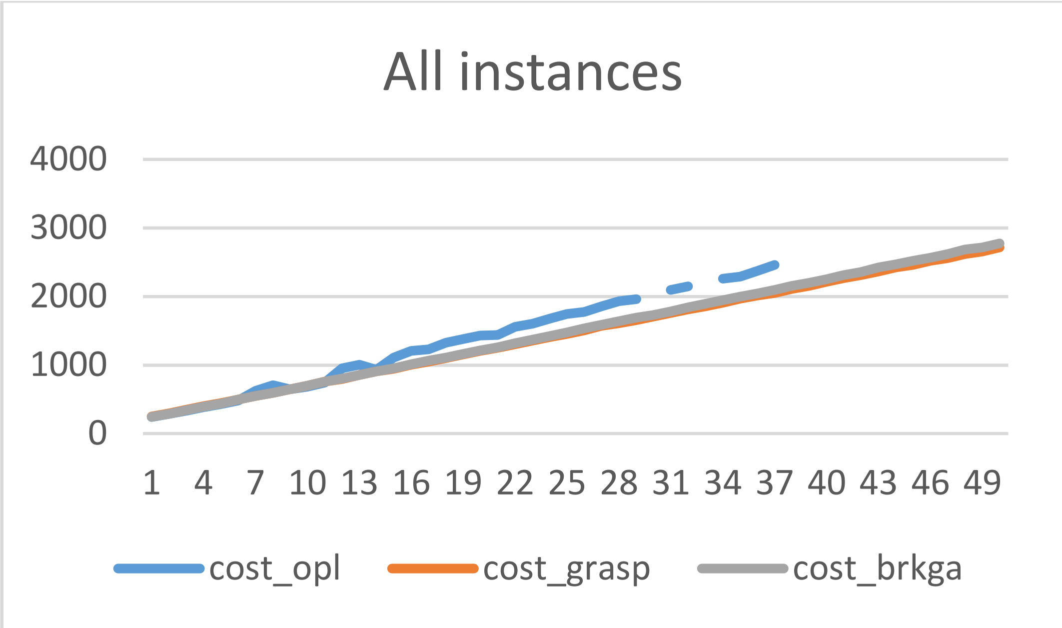


Figure 2: Cost of the solution found by the 3 solvers for all the 50 instances with maxTime = 10 min

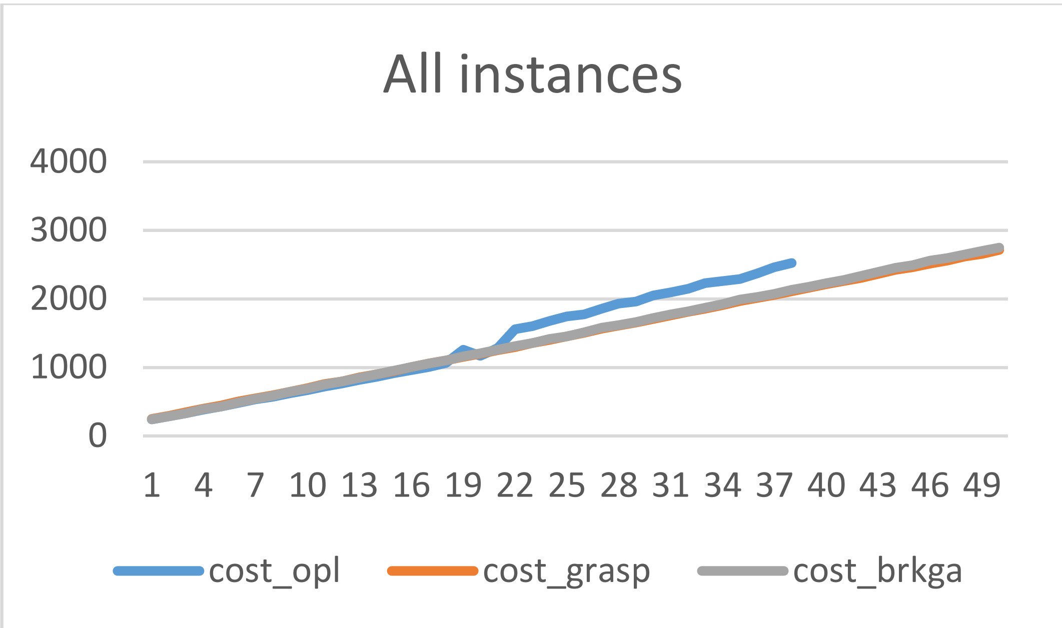


Figure 3: Cost of the solution found by the 3 solvers for all the 50 instances with maxTime = 20 min

6 Conclusions

In this paper we have seen three different approaches to solve an optimization combinatorial problem. Before making any strong conclusion, we should say that the performance of this approaches depends a lot on the implementation of the method for this specific problem. We cannot say in general that one algorithm is better than another.

For the **OPL** we have seen that is very sensitive to the formulation of the lineal model. Models that use less variables are much faster than other that use more variables. The strong point of this method is that, if it exists a solution it always finds the best one. If the algorithm finishes and it has not found any solution, that means that no solution exists. The problem is the explosion in the execution time for large instances, although it is quite fast for small instances.

In that problem, the **GRASP** meta-heuristic is the one that can solve large problems with little time. That is because it goes straightforward to a feasible solution. The problem is it doesn't improve or improves only a little bit the solution when it has more computational time. The better solution it can find is not very close to the optimal one.

The **BRKGA** has been the most difficult algorithm to program. It can go closer to the optimal solution, although, for large problems, it needs some time to reach a feasible solution. It is a very clever method. However, for that problem, it doesn't stand out, because OPL beats it for small problems and GRASP beats it for large problems.

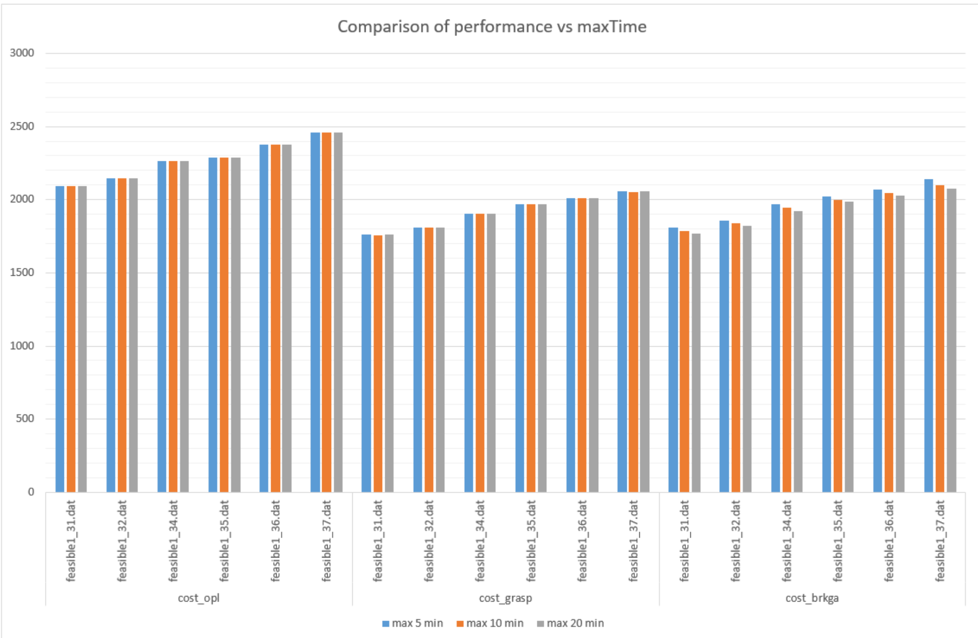


Figure 4: Comparison of the performance when we increase the execution time

7 Appendix

7.1 Data and plots of the results

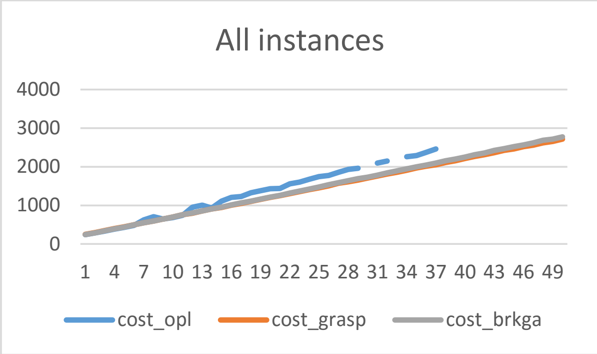
7.1.1 Results max 5 min

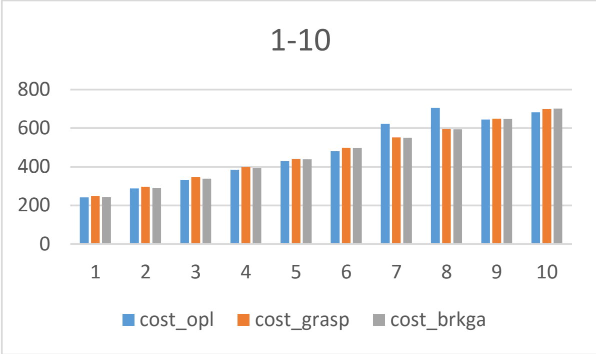
Max 5 min			
dat_file	cost_opl	cost_grasp	cost_brkga
feasible1_1.dat	241	250	245
feasible1_2.dat	288	297	293
feasible1_3.dat	332	344	345
feasible1_4.dat	385	400	401
feasible1_5.dat	521	444	443
feasible1_6.dat	526	502	499
feasible1_7.dat	656	551	556
feasible1_8.dat	717	598	601
feasible1_9.dat	645	649	655
feasible1_10.dat	830	699	705
feasible1_11.dat	908	755	765
feasible1_12.dat	969	798	803
feasible1_13.dat	1023	854	862
feasible1_14.dat	981	905	915
feasible1_15.dat	1111	949	964
feasible1_16.dat	1209	1005	1026
feasible1_17.dat	1227	1055	1077
feasible1_18.dat	1326	1103	1120
feasible1_19.dat	1375	1157	1177
feasible1_20.dat	1431	1203	1223
feasible1_21.dat	1437	1250	1285
feasible1_22.dat	-1	1304	1325
feasible1_23.dat	1599	1355	1384
feasible1_24.dat	1675	1406	1436
feasible1_25.dat	1742	1458	1495
feasible1_26.dat	1772	1506	1542
feasible1_27.dat	1858	1570	1603
feasible1_28.dat	1929	1611	1649
feasible1_29.dat	1963	1653	1709
feasible1_30.dat	-1	1709	1752
feasible1_31.dat	2096	1762	1811
feasible1_32.dat	2146	1810	1858
feasible1_33.dat	-1	1864	1910
feasible1_34.dat	2263	1906	1968
feasible1_35.dat	2286	1968	2025
feasible1_36.dat	2374	2012	2072
feasible1_37.dat	2460	2058	2138
feasible1_38.dat	-1	2108	2165
feasible1_39.dat	-1	2161	2230
feasible1_40.dat	-1	2213	2281
feasible1_41.dat	-1	2261	-1
feasible1_42.dat	-1	2314	-1
feasible1_43.dat	-1	2370	-1
feasible1_44.dat	-1	2425	-1
feasible1_45.dat	-1	2467	-1
feasible1_46.dat	-1	2516	-1
feasible1_47.dat	3075	2565	-1
feasible1_48.dat	-1	2624	-1
feasible1_49.dat	-1	2659	-1
feasible1_50.dat	3093	2721	-1

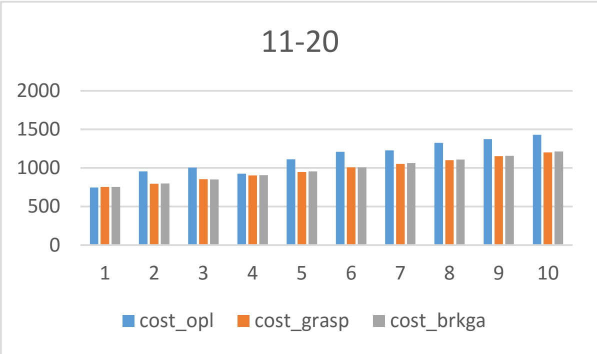

All instances
4000
3000
2000
1000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
cost_opl
cost_grasp
cost_brkga
1-10
1000
500
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga
11-20
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga
21-30
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga
31-40
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga
41-50
4000
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

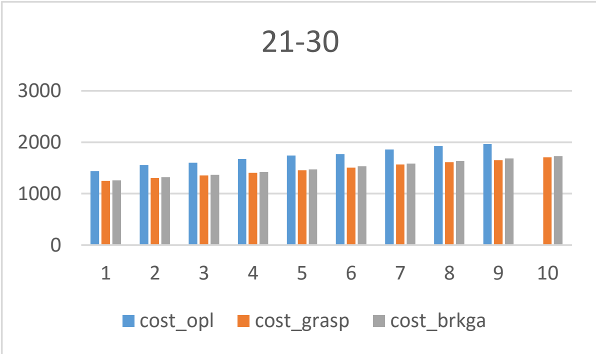
7.1.2 Results max 10 min

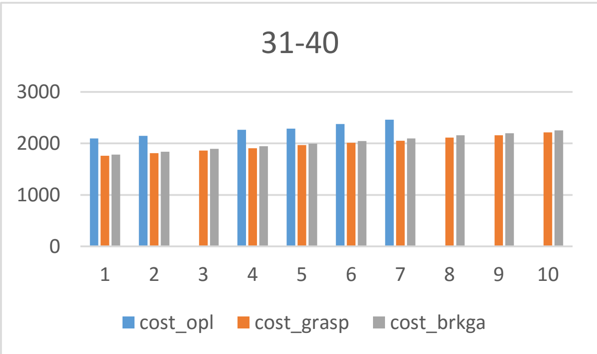
Max 10 min			
dat_file	cost_opl	cost_grasp	cost_brkga
feasible1_1.dat	241	249	243
feasible1_2.dat	288	296	290
feasible1_3.dat	332	346	338
feasible1_4.dat	385	399	392
feasible1_5.dat	429	442	438
feasible1_6.dat	481	498	497
feasible1_7.dat	623	552	550
feasible1_8.dat	704	596	594
feasible1_9.dat	644	649	647
feasible1_10.dat	682	698	701
feasible1_11.dat	746	754	755
feasible1_12.dat	955	796	799
feasible1_13.dat	1003	854	852
feasible1_14.dat	927	904	906
feasible1_15.dat	1111	948	955
feasible1_16.dat	1209	1006	1008
feasible1_17.dat	1227	1052	1064
feasible1_18.dat	1326	1100	1107
feasible1_19.dat	1375	1155	1158
feasible1_20.dat	1431	1203	1213
feasible1_21.dat	1437	1251	1262
feasible1_22.dat	1559	1302	1320
feasible1_23.dat	1599	1354	1367
feasible1_24.dat	1675	1405	1421
feasible1_25.dat	1742	1455	1474
feasible1_26.dat	1772	1507	1533
feasible1_27.dat	1858	1569	1584
feasible1_28.dat	1929	1612	1637
feasible1_29.dat	1963	1654	1688
feasible1_30.dat	-1	1708	1730
feasible1_31.dat	2096	1758	1784
feasible1_32.dat	2146	1809	1839
feasible1_33.dat	-1	1859	1895
feasible1_34.dat	2263	1907	1947
feasible1_35.dat	2286	1968	1997
feasible1_36.dat	2374	2010	2045
feasible1_37.dat	2460	2052	2097
feasible1_38.dat	-1	2112	2157
feasible1_39.dat	-1	2158	2199
feasible1_40.dat	-1	2212	2253
feasible1_41.dat	-1	2264	2309
feasible1_42.dat	-1	2312	2358
feasible1_43.dat	-1	2366	2423
feasible1_44.dat	-1	2423	2469
feasible1_45.dat	-1	2464	2520
feasible1_46.dat	-1	2518	2566
feasible1_47.dat	3075	2562	2616
feasible1_48.dat	-1	2620	2687
feasible1_49.dat	-1	2655	2719
feasible1_50.dat	3093	2716	2774

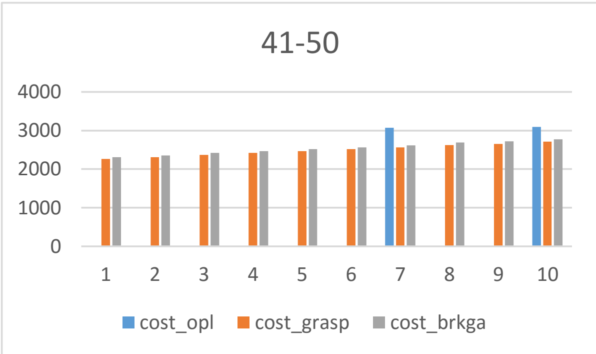

All instances
4000
3000
2000
1000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
cost_opl
cost_grasp
cost_brkga


1-10
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga


11-20
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

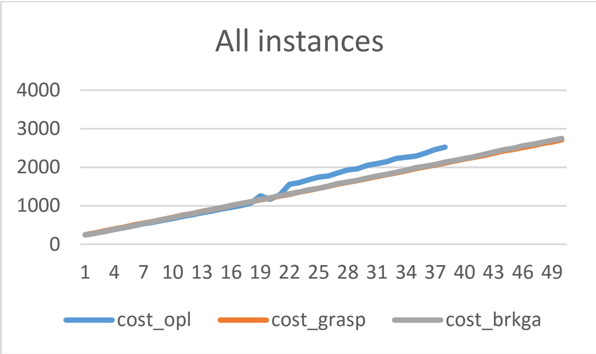

21-30
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

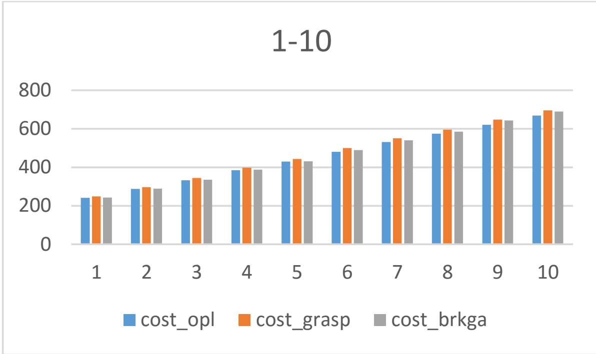

31-40
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

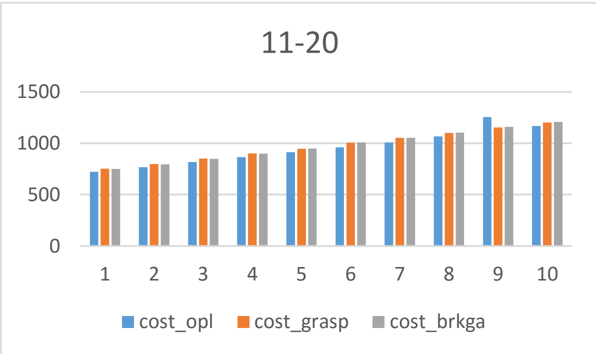

41-50
4000
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

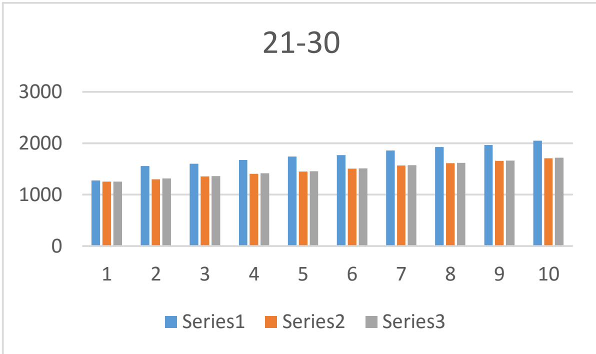
7.1.3 Results max 20 min

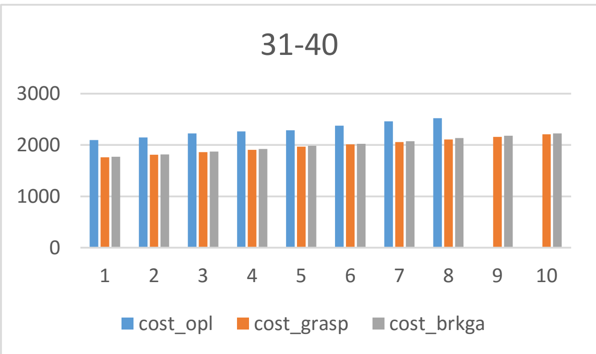
Max 20 min			
dat_file	cost_opl	cost_grasp	cost_brkga
feasible1_1.dat	241	249	243
feasible1_2.dat	288	297	289
feasible1_3.dat	332	344	335
feasible1_4.dat	385	398	388
feasible1_5.dat	429	443	431
feasible1_6.dat	481	500	489
feasible1_7.dat	531	551	540
feasible1_8.dat	574	595	585
feasible1_9.dat	621	648	643
feasible1_10.dat	668	696	689
feasible1_11.dat	722	754	750
feasible1_12.dat	766	797	794
feasible1_13.dat	817	852	849
feasible1_14.dat	864	902	899
feasible1_15.dat	912	947	950
feasible1_16.dat	959	1004	1007
feasible1_17.dat	1007	1054	1053
feasible1_18.dat	1067	1100	1103
feasible1_19.dat	1256	1155	1160
feasible1_20.dat	1168	1202	1207
feasible1_21.dat	1278	1253	1256
feasible1_22.dat	1559	1299	1314
feasible1_23.dat	1599	1355	1358
feasible1_24.dat	1675	1403	1415
feasible1_25.dat	1742	1452	1456
feasible1_26.dat	1772	1505	1510
feasible1_27.dat	1858	1567	1576
feasible1_28.dat	1929	1611	1616
feasible1_29.dat	1963	1655	1663
feasible1_30.dat	2051	1708	1720
feasible1_31.dat	2096	1761	1771
feasible1_32.dat	2146	1808	1819
feasible1_33.dat	2226	1859	1874
feasible1_34.dat	2263	1906	1923
feasible1_35.dat	2286	1968	1987
feasible1_36.dat	2374	2010	2026
feasible1_37.dat	2460	2059	2076
feasible1_38.dat	2523	2109	2134
feasible1_39.dat	-1	2159	2179
feasible1_40.dat	-1	2211	2227
feasible1_41.dat	-1	2262	2275
feasible1_42.dat	-1	2306	2334
feasible1_43.dat	-1	2363	2392
feasible1_44.dat	-1	2424	2452
feasible1_45.dat	-1	2465	2492
feasible1_46.dat	-1	2516	2557
feasible1_47.dat	3075	2556	2599
feasible1_48.dat	-1	2620	2648
feasible1_49.dat	-1	2659	2703
feasible1_50.dat	3093	2714	2748

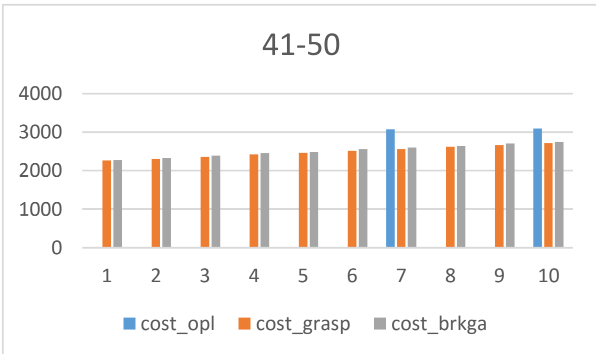

All instances
4000
3000
2000
1000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
cost_opl
cost_grasp
cost_brkga


1-10
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga


11-20
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga


21-30
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
Series1
Series2
Series3


31-40
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga


41-50
4000
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
cost_opl
cost_grasp
cost_brkga

7.2 The generators

Three different generators have been created. The first one produces a totally random instance. The last two produce always an instance that is feasible.

7.2.1 Random generator

The **random generator** takes the hours in a day and different random distributions for all the parameters that conform an instance, except for the number of nurses. The number of nurses is calculated adding a random positive integer to the maximum demand.

7.2.2 Feasible 1

The **feasible 1** takes the hours in a day and different random distributions for *maxHours*, *maxConsec*, *maxPresence* and the *demand*. Then it builds in a greedy way a solution that satisfies all the constraints. *minHours* is calculated as the minimum amount of hours that works the nurse that works less. The number of nurses is the number of nurses used by the generator to build its feasible solution.

7.2.3 Feasible 2

The **feasible 2** takes the hours in a day and a random distribution for the number of nurses. Then it randomly assign working hours to the nurses. *maxHours*, *minHours*, *maxConsec*, *maxPresence* and *demand* are calculated in the most restrictive way using the set of nurses.