Report of the AMMM Projet

Mathieu Chiavassa, Anthony Nixon
Master MIRI Data Science, FIB, BarcelonaTech, UPC

January 9, 2018

Contents

1 Problem statement **2**

2 Linear model for optimization in OPL **3**
2.1 Results for the OPL code 4

3 The BRKGA method **6**
3.1 Our decoder and fitness procedures 6
3.2 Results for BRKGA method 8
3.3 Conclusion for BRKGA method 11

4 The GRASP method **11**
4.1 Results for GRASP method 14
4.2 Conclusion for BRKGA method 15

5 Comparison of Results **16**

A Appendix : OPL script **17**

‘

1 Problem statement

The problem we have to solve is to find an optimal daily schedule for a given number of nurses in an hospital. The optimal workload is given by the hospital for all the hours of the day. The nurses schedule must satisfiy a list of constraints.
To model the problem, we introduce the following input parameters :

- $numNurses$: Number of nurses available
- $hours$: Total number of hours in one working day
- $demand$: Vector representing the number of nurses who are supposed to work at each hours
- $minHours$: Minimal number of working hours for a given nurse
- $maxHours$: Maximal number of working hours for a given nurse
- $maxConsec$: Maximal allowed number of consecutive working hours for a given nurse
- $maxPresence$: Maximal number of hours the nurses can spend at the hospital for a given nurse

With these parameters, we can introduce the different constraints :

C1 : For each hour h, at least, $demand[h]$ nurses should be working

C2 : Each nurse should work at least $minHours$ hours

C3 : Each nurse should work at most $maxHours$ hours

C4 : Each nurse should work at most $maxConsec$ consecutive hours

C5 : No nurse can stay at the hospital for more than $maxPresence$ hours

C6 : No nurse can rest for more than one consecutive hour

The project consists in solving this problem using three different optimization methods: the linear model with the OPL software and the two metaheuristic methods BRKGA and GRASP.

2 Linear model for optimization in OPL

In order to solve this optimization problem in OPL we used the aforementioned parameters but we also introduce the following ones :

- range N : Range between 1 and $numNurses$
- range H : Range between 1 and $hours$
- works[numNurses][hours] : Boolean matrix of size $numNurses * hours$.
  Tells for each nurse if he/she is working at a given hour,i.e, if $works[i][j] = 1$ then the nurse i is working during the hour j
- worksBefore[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- worksAfter[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- rests[numNurses][hours] : Boolean matrix of size $numNurses * hours$
- used[numNurses] : Boolean vector telling if a nurse is working during the day (value 1) or not (value 0).

The objective function to be minimized is therefore

$$\sum_{i=1}^{n} used[i].$$

The solution must satisfy the aforementioned constraints C1 to C6 we expressed in the following form to be used by the OPL code:

C1 : $\forall h, \sum_{i=1}^{n} works[i][h] \geq demand[h]$
For each hour, the sum of nurses working should be greater or equal than the number of nurses needed.

C2 : $\forall n, \sum_{i=1}^{h} works[n][i] \geq minHours * used[n]$
For each nurse, the sum of working hours in her schedule should be greater or equal than the minimum number of working hours.

C3 : $\forall n, \sum_{i=1}^{h} works[n][i] \leq maxHours * used[n]$
For each nurse, the sum of working hours in her schedule should be lower or equal than the maximum number of working hours.

C4 : $\forall n, \forall j \in [1; hours - maxConsec], \sum_{i=j}^{j+maxConsec} works[n][i] \leq maxConsec * used[n]$

C5 : $\forall n$, $\forall h$ if $h \leq hours - maxPresence$:
$worksBefore[n][h] + worksAfter[n][h + maxPresence] \leq 1$

C6 : $\forall n$, $\forall h \leq hours - 1$:
$worksAfter[n][h] \geq worksAfter[n][h + 1]$
$worksBefore[n][h] \leq worksBefore[n][h + 1]$
$rests[n][h] + rests[n][h + 1] \leq 1$
$rests[n][h] = (1 - works[n][h]) - (1 - worksAfter[n][h]) - (1 - worksBefore[n][h])$

The full OPL code in written the appendix A.

2.1 Results for the OPL code

In this section, we are going to present 3 tests based on 3 different datasets, a small one (TEST1), a medium one (TEST2), and a big one (TEST3). The different datasets are in the OPL data file.

TEST1:
In this test we try to optimize the schedule of 30 nurses for a 9 hours working day, with the following demand workload :
`demand = [ 5 3 8 5 1 7 5 6 2 ];`
And the following nurses parameters:
`numNurses = 30; hours = 9; minHours = 3; maxHours = 6; maxConsec = 7; maxPresence = 8;`

On this small dataset, OPL took less than 1 second to find the optimal solution. The objective function is equal to 8, which is the max of the demand vector. It means that not a single nurse is useless, and the solution is optimal.

TEST2:
In this test we try to optimize the schedule of 200 nurses for a 24 hours working day, with the following demand workload :
`demand = [ 53 24 33 40 70 12 33 55 66 12 30 22 55 77 88 22 34 55 22 55 23 22 11 12 ];`
And the following nurses parameters:
`numNurses = 200; hours = 24; minHours = 6; maxHours = 12; maxConsec = 6; maxPresence = 18;`

OPL took only 3 minutes to solve this problem. But the objective function value is 108. This solution is optimal although the scheduled nurses is higher than the max demand because the constraints limit the possible schedules for the nurses so we need more nurses to cover the range of demand hours.

TEST3:
In this test we try to optimize the schedule of 900 nurses for a 24 hours working day, with the following demand workload :

```
demand = [482 325 483 510 412 193 414 476 305 234 202 280 431 300 549 426
460 508 448 178 307 345 413 178];
```

And the following nurses parameters:

```
numNurses = 900; hours = 24; minHours = 6; maxHours = 18; maxConsec = 7;
maxPresence = 24;
```

Since it is a huge dataset, OPL took 21 minutes to optimize this problem. But the result is optimal, the objective function is equal to 549 nurses. Again, it's higher than the max of demand hours for the same reason as the previous test.

3 The BRKGA method

In this section, we use a Biased Random Key Genetic Algorithm (BRKGA), to solve the nurses scheduling problem. This method belongs to the genetic algorithms family and has been introduced by Bean (1994) and developed later for solving large combinatorial optimization problems. The basic principle is to mimic the evolution of individuals in a population submitted to natural selection. Basically, for a given criteria, called *fitness*, the best individuals are selected and are crossed with the other individuals of the population. Generations after generations, the individuals that optimize the fitness are selected.
We based our work in this section on the paper of José Fernando Goncalves and Mauricio G. C. Resende, *Biased Random-Key Genetic Algorithm for Combinatorial Optimization* and on the python code provided by profesor M. Ruiz Ramirez.

Each individual of the population is represented by a *chromosome* and corresponds to a candidate for the solution we are looking for. The chromosome consists of a given number of *genes* that takes a real value in the interval $[0, 1]$. The gene are used to compute the fitness of the individual which is correlated to the problem we want to optimize. To solve a minimization problem such the nurse scheduling described in section 1, the BRKGA used the following steps:

1 define the size p of the population, and the number of generations

2 initialize randomly the genes of the p individuals

3 decode the genes to compute the fitness of each individual

4 select the best individuals, *elite*, and generate the next generation chromosomes

5 go to step 3 until the number of generations is not reached

6 decode the best fitness chromosome of the last generation

If the problem and the parameters have been chosen correctly, the individual with the best fitness in the last generation is a good candidate for the optimum of our problem.

The tricky part of the BRKGA algorithms is the decoder procedure and a huge number of research papers deal with this problem. If the chromosome is composed of N genes, the decoder must map the hypercube space $[0, 1]^N$ to the solution space. A suited fitness has also to be defined o select the best candidate in the solution space.

3.1 Our decoder and fitness procedures

The decoder procedure we propose will map the chromosome to the daily schedule for all the nurses, represented by the boolean array $works$ of size $numNnurse \times hours$ introduced in section 2. The chromosome length is therefore $numNurse \times hours$. The

mapping from floating genes value to boolean ones is simply done by the transform procedure:

Procedure transform

```
for all the genes of the population
  if gene_value ≤ 1-minHours/hours then solution_value=0
  else solution_value=1
```

With this procedure, the probability of working for a nurse depends on the percentage she has to work during the day. If the ratio is small, then the transform procedure will produce more 0 values (not work). On the contrary more values 1 will be produced. (This improves significantly the results in our experiments comparing to use the fixed value 0.5).

Once we have the daily schedule, we will check if the schedule satisfies all the constraints of the problem described in section 1 and define a corresponding value of the fitness. First for a given nurse, its schedule is verified:

Procedure checkschedule

```
test=0
  check all the constraints from C2 to C6 defined in section 1
  for each non satisfied constraint do test += 1
return test
```

The way we check in practice the constraints is detailled in the function `checkschedule` in DECODER_DUMMY.py. The value of test will be used for the fitness.

We then compare the workload of the day with the theoretical demand specified by the user, in order to check if the constraint C1 is satisfied:

Procedure compareworkload

```
error=0
for all the hours h of the day
 compute the number of working nurses:  workload(h)
 if workload(h) > demand(h) then error += workload(h)-demand(h)
 if workload(h) < demand(h) then error += 10*|workload(h)-demand(h)|
return error
```

For this error we penalize 10 times more the fact that the workload is less than the demand. When the error is larger than the demand, the optimal soltuion is not reached, but the solution is acceptable in practice. If the workload is exactly aquals to the demand, the error is 0.

We can then define the complete decode procedure

Procedure decode

```
transform
for all the individuals in the population
  fitness=0
  for all the nurses
    test=checkschedule
    fitness += 10.*test/numNurses
  error=compareworkload
  fitness += error/hours
```

With this definition of the fitness, we have the following properties:

- if all the nurses schedules are ok and if the workload is exactly the same than the demand, then the fitness is equal to 0
- when a nurse schedule is not satisfactory, the fitness is increased by an amount of $10 * test/numNurses$. It means that the penalization depends on the number of constraints that are not satisfied. The factor 10 has been found from the experiments and is necessary to gives more weight to the schedule part of the fitness than to the error one. In practice it means that obtaining correct schedules is important! We also normalize by the number of nurses to have a criteria mostly independent of the data.
- when the workload is not optimal, the fitness is increased depending of the error normalized by the number of hours in the day. Due to the definition of the error function, the fitness is highly penalized for the hours where the workload is less than the demand.

Since the python BRKGA procedure proposed by in this project is written to minimize the fitness, we use directly this decoding procedure in the full BRKGA python code.

The python scripts of all our procedures are given in the attached file with the report and are written in the file DECODER_DUMMY.py. The specific data are in the DATA_DUMMY file.

3.2 Results for BRKGA method

We present three tests in this section, a simple one (TEST1), a medium one (TEST2) and un more complex (TEST3). The data corresponding to these tests are in the file DATA_DUMMY. All the tests are performed on a MacBook with Intel Core i7 at 2.7 GHz.

TEST1:
This test optimize the schedule of 10 nurses for a 12 hours working day. After some run, we obtain an almost perfect schedule using the following parameters:
`'numIndividuals': 150, 'maxNumGen':700, 'eliteProp':0.1, 'mutantProp':0.25, 'inheritanceProb':0.65`
`'nNurses': 10, 'nHours': 12, 'minHours':3, 'maxHours':7, 'maxPresence':10, 'maxConsec':4`
And with the following workload demand:
`'demand': [9,8,8,10,11,12,9,10,16,11,10,14,13,13,9,15,12,10]`
We obtain the optimal solution: all the schedules of the nurses satisfy the constraints, the demand is exactly satisfied. The corresponding fitness is represented on figure 1 and reaches the 0 value after 60 generations.

```
Nurse 0 Test Schedule = 0
Nurse 1 Test Schedule = 0
Nurse 2 Test Schedule = 0
Nurse 3 Test Schedule = 0
Nurse 4 Test Schedule = 0
Nurse 5 Test Schedule = 0
Nurse 6 Test Schedule = 0
Nurse 7 Test Schedule = 0
Nurse 8 Test Schedule = 0
Nurse 9 Test Schedule = 0
Comp.  Workload [0, 2, 4, 5, 2, 7, 6, 4, 3, 3, 1, 1]
Dema.  Workload [0, 2, 4, 5, 2, 7, 6, 4, 3, 3, 1, 1]
Total number of working nurses= 10
```

This solution is obtain in less than 10 seconds. For all the runs we made for this TEST1, we mostly obtain a zero fitness solution in less than 150 generations. In the case where the fitness is not 0, the schedules are always correct but the workload is overestimated since too much nurses are working comparing to the demand. This corresponds to an acceptable solution, but not the optimal one.

TEST2:
In this test we try to optimize the schedule of 30 nurses for a 9 hours working day where the demand workload is :
`'demand': [5, 3, 8, 5, 1, 7, 5, 6, 2]`
and the nurses parameters:
`'nNurses': 25, 'nHours': 18, 'minHours':4, 'maxHours':8, 'maxPresence':10, 'maxConsec':5`
In that case, we have 450 genes for each individual of the population.
And the following genetic parameters:
`'numIndividuals': 250, 'maxNumGen':800, 'eliteProp':0.1, 'mutantProp':0.3, 'inheritanceProb':0.6`

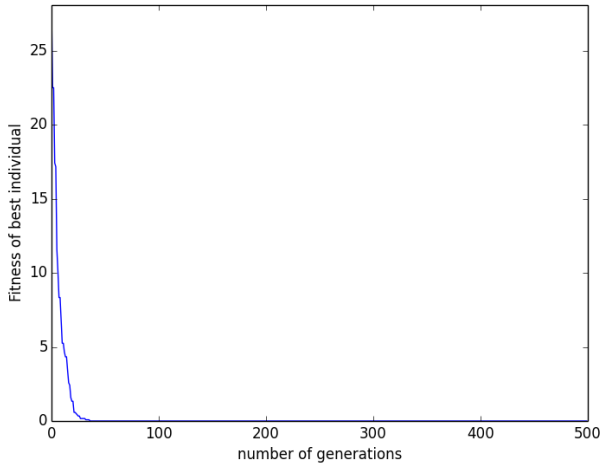

Figure 1: Evolution of the fitness for the TEST1.

We obtain a good solution in less than 2 minutes with 17 nurses, all the nurses passed all the constraints. The only bad point is that the workload is a bit overestimated compare to the demand vector.
The corresponding fitness is represented on figure 2 and get stuck at 1 after 100 generations. Since all the nurses are good, it means it can't reduce the gap between the demand and the actual demand workload.

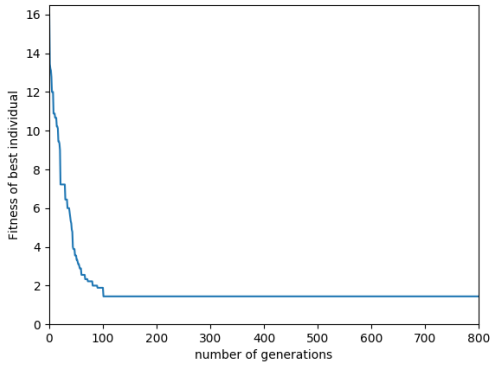

Figure 2: Evolution of the fitness for the TEST2.

We perform a huge number of numerical experiments exploring the parameters space to try to obtain the optimal solution. Unfortunately, we never obtain this optimal solution!
Adding more individuals in the population do not always results in better results, but of course it significantly increases the cpu time!

TEST3:
Here we try to optimize the schedule of 200 nurses for a 24 hours working day with this workload demand:
`[53, 24, 33, 40, 70, 12, 33, 55, 66, 12, 30, 22, 55, 77, 88, 22, 34, 55, 22, 55, 23, 22, 11, 12]`
The following nurses parameters:
`'nNurses': 200, 'nHours':24, 'minHours': 6, 'maxHours': 12, 'maxConsec': 6, 'maxPresence': 18` And the following genetic parameters :
`'numIndividuals': 250, 'maxNumGen':800, 'eliteProp':0.1, 'mutantProp':0.3, 'inheritanceProb':0.6`

We obtained a solution but it's very bad, some nurses failed one or more constraints and at some hours we have a huge gap between the real demand and the actual workoad. We tried to change the genetic parameters, for example using a bigger proportion of elite or more mutations, but the result didn't really get better. We also tried to run this dataset with more individuals and more generations but it was too heavy for the computer and after few crashes we decided to stop.

3.3 Conclusion for BRKGA method

We found this method very interesting since it is based on very simple concepts to optimize a complex problem. Our version gives most of the time the optimal solution for problems with less than about 350 genes. Nevertheless, we are a little bit disappointed since we do not obtain correct results for larger problems (i.e the TEST3), despite many efforts...
Some improvements could be done, adding also a mutation procedure of the chromosomes, or increasing the number of mutants, and the probability of crossover as we have seen in many AG papers. It will may be help to get out of the local minima by modifying randomly the genes of the individuals.
But the best improvements would be to define a decoding procedure that gives directly a correct schedule for all the nurses. In that case, only the workload would be optimized in a very smaller space. Unfortunately, we did not found this procedure, but we will be very interesting if it exists!

4 The GRASP method

The greedy randomized adaptive search procedures method (GRASP) has a constuction phase and a local search phase. Iteration through these phases obtains a good, but not necessarily optimal, solution by creating feasible solutions then refining them during each iteration. In our project the following algorithms were implemented:

Procedure construct(α, C = candidate_set)

```
possible_solution = ∅
solved = False
Initialize demand vector D⃗
for k = 1,2,...,numNurses do
  s_min = min{g(t, D⃗) | t ∈ C}
  s_max = max{g(t, D⃗) | t ∈ C}
  RCL = {s ∈ C | g(s, D⃗) ≤ s_min + α * (s_max − s_min)}
  select random s from RCL
  possible_solution = possible_solution ∪ s
  Subtract s from D⃗
  if max(D⃗) is 0 do
    solved = True
    break
  end if
end for
if solved is True do
  return possible_solution
else infeasible
```

The constuct function builds up a feasible solution (one where all the demand is satisfied by a subset of available nurses with valid schedules) by adding one nurse at a time until all the demand is satisfied or the pool of nurses is exhausted. A solution is a matrix of dimension numNurses x numHours where each row is a binary vector representing a nurse schedule where 1 is a worked hour and 0 is unworked. Many GRASP implementations will update the candidate list after each addition, however, in our problem nurses are independent of each other and the schedule of one nurse has no effect on the constraints of the next nurse. We only need to update the demand. The candidate list of possible schedules is preconstructed against the constraints and passed in as an argument.

To decide which schedule to assign each nurse added to the solution we score how each schedule in the candidate list will affect the demand vector via our **greedy function g()** described below. After scores are assigned to candidates we create a restricted candidate list which creates a sub-list of best candidates based on the **RCL Equation**: $s_{min} + \alpha * (s_{max} - s_{min})$. α is a value [0-1] which defines a percentage of best scored candidates. A value of 0 means it's most greedy and will always pick the best,

a value of 1 means highest randomness and any candidate might be added to solution at that stage. The point of the RCL is to introduce an element of randomness to the constructed solution to get new local search minimums.

Procedure g(schedule, $\vec{D}$ = demand)

```
1 score = 0
2 maxd = max(D⃗)
3 for k = 1,2,...,numNurses do
4   Δ = D⃗[k] - schedule[k]
5   if Δ not 0 do
6    score = score + exp^(delta/maxd)
7   end if
8 end for
9 return score
```

Our **GREEDY function g()** uses the exponential function to accumulate a score on how effective the schedule at satisfying demand. If the schedule was a perfect match ($\Delta = 0$) no penalty score would be added. We chose exponential because it accumulates penalties for both failing to staff a demand hour and for overstaffing a demand hour, the case when Δ of a given hour is 1 or -1 respectively. Lastly, we normalized the function because of buffer limitations in runtime.

Procedure local_search(x = feasible_solution)

```
1 neighbor_set = {y ∈ N(x) | f(y) < f(x)}
2 best_neighbor = x
3 while neighbor_set is ≠ ∅ do
4   select random n from neighbor_set
5   best_neighbor = n
6   neighbor_set = {y ∈ N(best_neighbor) | f(y) < f(best_neighbor)}
7 end while
8 return best_neighbor
```

Local search refines the initial feasible solution but generating a set of nearby solutions defined by the neighbor function N() (described below), which improve the objective function f() (described below). From this set the algorithm descends to a local min by choosing a random solution and then repeating the process (finding it's neighbor set and comparing their f() values), until we are left with a single element which has no better neighbors.

Procedure f(solution)

```
1 return number of nurses in solution (count rows in solution)
```

Objective function f() returns the number of nurses used in a solution by returning the number of rows, each of which represent one nurse working in the solution.

Procedure N(x = solution)

```
1 neighbor_set = ∅
2 Initialize demand vector D⃗
3 nurses_working = {(t_1, t_2, ..., t_n) | ∀j ∈ {1, ..., nHours}, t_j = Σ_{i=1}^{numNurses} x_ij}
4 demand⃗_Δ = nurses_working⃗ - D⃗
5 for each row in solution do
6   Initialize neighbor_sol to solution
7   row_effect⃗ = demand⃗_Δ − row⃗
8   negative_count = number of negatives in row_effect⃗
9   if negative_count is 0 do
10    remove row from neighbor_sol
11    add neighbor_sol to neighbor_set
12  end if
13 end for
14 return neighbor_set
```

The neighbor function N() returns a set of all neighbors to the input solution. It defines a neighbor to be any solution that still satisfies demand after removing a nurse from the original solution. To implement this we tracked demand surpluses from the original solution in the $\vec{demand}\_\Delta$ variable and then simulated how this overall demand remained satisfied after each possible row removal via $\vec{row\_effect}$. A negative value in this vector indicates that there is demand that is unsatisfied.

4.1 Results for GRASP method

We performed numerous runs on the following test sets

Small problem: nNurses: 30, nHours:9, minHours: 3, maxHours: 6, maxConsec: 7, maxPresence: 8, demand: [5, 3, 8, 5, 1, 7, 5, 6, 2]

Sample Small solution (maxITR=10, alpha=0.35):
Number Nurses: 9
Totals: [5, 3, 9, 6, 6, 8, 5, 6, 3]
Demand: [5, 3, 8, 5, 1, 7, 5, 6, 2]
Overstaff: [0, 0, 1, 1, 5, 1, 0, 0, 1]
Time: 0.0535528659821

Sample Small solution (maxITR=5, alpha=0.1):
Number Nurses: 8
Totals: [5, 3, 8, 5, 6, 7, 5, 6, 2]
Demand: [5, 3, 8, 5, 1, 7, 5, 6, 2]
Diff: [0, 0, 0, 0, 5, 0, 0, 0, 0]
Time: 0.0300381183624

Mid problem: nNurses: 200, nHours:24, minHours: 6, maxHours: 12, maxConsec: 6, maxPresence: 18, demand: [53, 24, 33, 40, 70, 12, 33, 55, 66, 12, 30, 22, 55, 77, 88, 22,

34, 55, 22, 55, 23, 22, 11, 12]

Sample Mid solution (maxITR=10, alpha=0.35):
Number Nurses: 125
Totals: [53, 25, 47, 46, 70, 63, 70, 78, 73, 81, 79, 80, 74, 85, 91, 69, 59, 63, 29, 55, 26, 29, 14, 12]
Demand: [53, 24, 33, 40, 70, 12, 33, 55, 66, 12, 30, 22, 55, 77, 88, 22, 34, 55, 22, 55, 23, 22, 11, 12]
Overstaff: [0, 1, 14, 6, 0, 51, 37, 23, 7, 69, 49, 58, 19, 8, 3, 47, 25, 8, 7, 0, 3, 7, 3, 0]
Time: 427.994287014

Sample Mid solution (maxITR=5, alpha=0.1):
Number Nurses: 123
Totals: [53, 24, 44, 42, 71, 52, 69, 55, 84, 62, 83, 76, 71, 90, 95, 70, 70, 55, 37, 55, 23, 22, 13, 12]
Demand: [53, 24, 33, 40, 70, 12, 33, 55, 66, 12, 30, 22, 55, 77, 88, 22, 34, 55, 22, 55, 23, 22, 11, 12]
Diff: [0, 0, 11, 2, 1, 40, 36, 0, 18, 50, 53, 54, 16, 13, 7, 48, 36, 0, 15, 0, 0, 0, 2, 0]
Time: 298.432577848

Large problem: nNurses: 900, nHours:24, minHours: 6, maxHours: 18, maxConsec: 7, maxPresence: 24, demand: [482, 325, 483, 510, 412, 193, 414, 476, 305, 234, 202, 280, 431, 300, 549, 426, 460, 508, 448, 178, 307, 345, 413, 178]

Sample Large solution (maxITR=10, alpha=0.35):
Unable to complete in reasonable time.

4.2 Conclusion for BRKGA method

The GRASP algorithm performed quite well and was straight forward to implement. Adjusting the parameters showed that raising the number of iterations (maxITR) brought solutions closer to optimal as well as lowering the alpha. In this particular problem the nurse schedules are disconnected/independent so in this case having an alpha of 0 and placing the best scoring schedule directly to the solution and bypassing the RCL would be a good strategy to create optimal solutions. However, in the general case of applying GRASP to optimization problems, the previously added solution component would often affects the candidate list available for the next entry. When ordering matters in this sense, then the alpha to give many permutations of feasible solutions to local search should be exceptional better at converging on a good solution. Future improvements to try on our implementation would be to allow local search to 'break' solutions and then rebuild by altering remaining schedules. This would help us get much closer to optimal solutions.

5 Comparison of Results

Comparing ILP to Metaheuristics

The ILP model in OPL guarantees an optimal solution whereas the metaheauristics only generate very good approximation which may be optimal, but are generally considered 'good enough'. Computationally, the ILP is locked to an exponential increase in

computational time related to the complexity of the problem. For very large data sets it is impractical. The metaheuristics can by-pass this limitation by taking short cuts, using rapid descents, and ramdomness to gain ever better solutions in a quicker amount of time. One downside to the metaheuristics are their sensitivity to input parameters which control their behavior.

In our experiments our implementations of the metaheuristics performed very well for very small datasets and outperformed the OPL, but for larger sets we actually performed a bit worse than OPL. This is not due to the metaheustics paradigms themselves but rather a combination of parameterization and also the efficiency of our fitness and scoring strategies which could be improved to descend to minima and a faster rate.

Comparing BRKGA and GRASP

Between the BRKGA and GRASP implementations our GRASP had a bit stronger performance. Our BRKGA was able to generate solutions for up to the MID size data set but the result was worse than we would expect. The GRASP was much faster and the tuning much simpler. We were able to generate good solutions for the MID data consistently. Both of our algorithms were far off optimal on the LARGE set and to modify the parameters to complete in reasonable time generated somewhat trivial solutions. However, our test hardware was modest and with an increase in processing power or ample time the metaheuristics will create useful solutions.

A Appendix : OPL script

OPL optimization code :

```
int numNurses = ...;
int hours = ...;
range N = 1..numNurses;
range H = 1..hours;

int demand [h in H]= ...;
int minHours = ...;
int maxHours = ...;
int maxConsec = ...;

int maxPresence = ...;

dvar boolean works[n in N][h in H];        // Whether nurse n works at hour h
dvar boolean worksBefore[n in N][h in H]; // Relative to a given hour, tracks if the nurse works before
dvar boolean worksAfter[n in N][h in H];  // Relative to a given hour, tracks if the nurse works after
dvar boolean rests[n in N][h in H];        // Whether a nurse rests at hour h
dvar boolean used[n in N];                                  // If nurse n is used or not

minimize sum(n in N) used[n];                      // Objective is to minimize the number of nurses
subject to {

        // Constraint 1
        // The number of provided nurses must be greater or equal to the demand of each hour
        forall(h in H)
                sum(n in N) works[n][h] >= demand[h];

        // Constraint 2
        // Each nurse that is working must work at least minHours.
        forall(n in N)
                sum (h in H) works[n][h] >= minHours*used[n];

        // Constraint 3
        // No nurse that is working can work more than maxHours hours.
        forall(n in N)
                sum (h in H) works[n][h] <= maxHours*used[n];

        // Constraint 4
        // Each nurse should work at most maxConsec consecutive hours - must have a break to work more.
        // - algorithm uses sliding window comparisons
        forall(n in N)
                forall(i in 1..(hours-maxConsec))
                        sum(j in i..(i+maxConsec)) works[n][j] <= maxConsec*used[n];

        // Constraint 5 and 6
        // No nurse can stay at the hospital for more than maxPresence number of hours
        // Rests must be no longer than one hour in length
        forall(n in N)
                forall (h in H: h <= hours-maxPresence)
                        worksBefore[n][h] + worksAfter[n][h+maxPresence] <= 1;

        forall(n in N)
                forall (h in H: h <= hours-1){
                        worksAfter[n][h] >= worksAfter[n][h+1];   // allowed: 11110, rejected: 10101
                        worksBefore[n][h] <= worksBefore[n][h+1]; // allowed: 00111, rejected: 01110
                        rests[n][h] + rests[n][h+1] <= 1;         // allowed: 10100, rejected: 11001
                }

        forall(n in N)
                forall (h in H)
                        rests[n][h] == (1-works[n][h]) - (1-worksAfter[n][h]) - (1-worksBefore[n][h]);
}
```