Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Package placement in trucks. An analysis comparison of combinatorial problem in cplex, GRASP and BRKGA metaheuristics

Author: **Alejandro Montero Rivero**

FIB – Master in Innovation and Research in Informatics

Spring Semester 2016

Index

0. Introduction

Large problems are common occurrences in industry, science, economics or government. This problems have in common a finite ground set X = {1,…,n}, and a function $f: X \rightarrow \mathbb{R}$ and one seeks a solution $x^* \in X$ such as $f(x^*) \leq f(x)\ \forall x \in X$. It is possible to find a solution to a combinatorial problem by evaluating all possible solutions and selecting the one closer to the stated objective. However this is not a feasible way to proceed in big problems, because the number of combinations often grows exponentially.

Since the last few decades, a lot of work has been done to address this issue, seeking new methods that do not explicitly require the examination of every single combination. Integer programming, one proposed solution, is a mathematical optimization in which some or all variables are restricted to be integers. These algorithms minimize or maximize a linear function, subject to equality, inequality and integer constraints. This enables accurate modeling of problems involving discrete quantities (such as stocks in a shop) or yes/no decisions.

Another approach is to use metaheuristics, a procedure designed to find, generate or select a solution that may provide a sufficiently good solution. There are lots of different approaches to metaheuristics; some algorithms try to create a local search strategy like “hill climbing” to find local optimums. Others try to mimic the nature by mutating and creating inheritance to create new solutions.

In this document we are introducing a combinatorial problem: our enterprise wants to place a certain number of packets in different trucks. To reduce the costs of the expedition we want to use as less trucks as possible to save gas, at the same time, it is well known that a vehicle with higher load consumes more so, in consequence, we want to minimize the amount of load of the highest loaded truck of the expedition.

To solve this problem, we are going to use and compare the performance of Cplex, a solution from IBM that implements integer programing and two different heuristic procedures: Greedy Randomized Adaptive Search Procedure (GRASP) and Biased Random-Key Genetic Algorithm (BRKGA).

1. The problem

In this section we are going to discuss the colloquial interpretation of the problem.

Objective function

The objective of the problem is to minimize the number of used trucks and the max loaded truck in the expedition. A used truck is a vehicle that is carrying at least one package. Every package has different weight that is not relative to the volume it occupies thus, a truck can be highly loaded, unbalancing the charge across all the expedition. In consequence, we want to minimize the load of the highest loaded truck as well.

Provided data

As in real case scenario, all our packages have different area and weight, while our trucks share the same dimensions and capacity across the expedition. At the same time for security reasons, some packages cannot be placed in the same trucks as another. Thus, as the managers of the expedition we have all the information we need to place all the packages in the trucks:

- **Number of trucks to available**.
- **Number of packets to place**.
- **Width of the trucks** (all trucks share the same width).
- **Height of the trucks** (all trucks share the same height).
- **Weight of the trucks** (all trucks share same weight)
- **Width of all the packages**.
- **Height of all the packages.**
- **Packages that are incompatible.**

2. Work methodology and Hardware

As in any project a certain work methodology has to be used to assure the correctness of the results, as well as to optimize time and resources. In this section we are going to discuss the work methodology as well as the hardware resources used in the execution.

The hardware used consists in laptop with the following hardware:

- Intel i7 4700U. 2.1 GHz up to 3.3 GHz in turbo. **2 cores, 4 threads**. 4MB cache.
- 8 GB RAM DDR3.
- 256 GB SSD.
-

First and one of the most important techniques in the science community, all executions for both Cplex and metaheuristics has been performed three times in a row, then the mean value is calculated and all the possible outliers are rejected so they do not pollute the sample.

For every Cplex execution the computer is left with no user's process working on background. Even though the average usage of the overall CPU is around a 30% a user actively using the computer may lead to interference with the performance.

As metaheuristics is executed in one single thread, Cplex is bounded to a single thread too. IBM's program offers high parallelism capabilities, but using the four available threads in the machine would not be a fair comparison between the two approaches.

Continuing with metaheuristics, the proper values for Alpha value in GRASP and Individuals in BRKGA are calculated using the problems shared in the comparison Cplex versus metaheuristics; as the optimization value is well known, the best value for the tuning options is easily located. In the analysis of GRASP versus BRKGA

3. Linear Programming

In this section we are going to introduce the formal Integer Linear Programming including all the input data, decision variables and a short description of and mathematical definition of the objective function and each constraint

Input Data

For this problem we have the following provided data:

- **nTrucks**: Total number of trucks.
- **nPackages**: Total number of packages.
- **xTruck**: Width of the trucks.
- **yTruck:** Height of the truck.
- **wTruck:** Weigt**h** of the truck.
- **xDimp**: Vector of size (nPackages) containing the width of all packages.
- **yDimp**: Vector of size (nPackages) containing the height of all packages.
- **wp:** Vector of size (nPackages) containing the weight of all packages.
- **Incomp**: Matrix of size (nPackages * nPacakges) containing the incompatibility value between each pair of packages.

Ranges

- **Range X: from 1 to xTruck**. Contains all the possible values for the width of the trucks.
- **Range Y: from 1 to yTruck**. Contains all the possible values for the height of the trucks.
- **Range T: from 1 to nTrucks**. Contains all the possible ID of the trucks.
- **Range P from 1 to nPacakges.** Contains all the possible ID of the packages.

Decision variables

A decision variable is an unknown value in an optimization problem. It has a domain which is a compact representation of the set of all possible values for the variable. The purpose of OPL is to find the values for the decision variables such as all constraints are satisfied and optimize the objective function. The following are the decision variables used in the problem:

- **Boolean $\boldsymbol{pt_{p,t}}$:** Boolean variable specifying if package p is being used in truck t.
- **Boolean $\boldsymbol{pbl_{p,x,y}}$**: Boolean variable specifying if the cell [x,y] contains the bottom-left cell of package p.
- **Boolean $\boldsymbol{pxy_{p,x,y}}$**: Boolean variable specifying if the cell [x,y] is occupied by the package p.
- **Boolean $\boldsymbol{used_t}$**: Boolean variable specifying if truck t is used in the expedition.

Objective Function

The objective function is expressed as the minimization of the highest loaded truck added to the number of used trucks multiplied by the weight of the truck.:

$$minimize\ z + \sum_{t=1}^{nTrucks} used[t] * wTruck$$

Constraints

(1)
$$z \geq \sum_{p=1}^{nPackages} pt[p][t] * wp[p]$$
$$\forall t \in nTrucks$$

(2)
$$\sum_{t=1}^{nTrucks} pt[p][t] == 1$$
$$\forall p \in nPackages$$

(3)
$$pt[p][t] \leq used[t]$$
$$\forall t \in nTrucks$$
$$\forall\, p \in nPackages$$

(4)
$$pt[p1][t] + pt[p2][t] + incomp[p1][p2] \leq 2$$
$$\forall t \in nTrucks$$
$$\forall\, p1 \in nPackages$$
$$\forall\, p2 \in nPackages$$

(5)
$$\sum_{p=1}^{nPackages} wp[p] * pt[p][t] \leq wTruck$$
$$\forall t \in nTrucks$$

(6)
$$\sum_{x=1}^{X} \sum_{y=1}^{Y} pbl[p][x][y] == 1$$
$$\forall p \in nPackages$$

(7)
$$\sum_{x \leq xTruck}^{X} \sum_{-xDimp[p]+1}^{y \leq yTruck} \sum_{-yDimp[p]+1}^{Y} pbl[p][x][y] == 1$$
$$\sum_{x > xTruck}^{X} \sum_{-xDimp[p]+1}^{y > yTruck} \sum_{-yDimp[p]+1}^{Y} pbl[p][x][y] == 0$$
$$\forall p \in nPackages$$

(8) $$\sum_{x-xDimp[p]< x2\leq x}^{X} \sum_{y-yDimp[p]<y2\leq y}^{Y} pbl[p][x2][y2] == pxy[p][x][y]$$

$$\forall p \in nPackages$$
$$\forall x \in X$$
$$\forall y \in Y$$

$$pxy[p1][x][y] + pt[p1][t] + pxy[p2][x][y] + pt[p2][t] \leq 3 \; if \; (p1 != p2)$$

(9)

$$\forall t \in nTrucks$$
$$\forall x \in X$$
$$\forall y \in Y$$
$$\forall p1 \in nPackages$$
$$\forall p2 \in nPackages$$

Explanation of the constraints

(1). This constraint complements the objective function making sure z has the value of the highest loaded truck of the expedition.

(2). These constraint specifies that all packages have to be loaded in just a single truck.

(3). A package cannot be placed in a non-used truck.

(4). Two packages cannot be placed in the same truck if they are incompatible.

(5). The weight of all packages inside a truck t cannot exceed the total capacity of the truck.

(6). A package can have only have one bottom cell position placed in a truck.

(7). This single constraint specifies the positions in which the pbl of a packet can be set to 1, or to be more clear, those positions in which the packet fits the truck. The firsts summations make sure the area inside the fitting area of the truck has exactly one position set to 1 while the second ones assure non plb is set to one. This constraint is especially important for two reasons: first of all, it limits the area in which a packet can be positioned, giving the solver different positions to place the package. Secondly, it assures a package will only be positioned in a place it actually fits the truck area.

(8). This constraint sets all the positions occupied by the packet (pxy[p][x][y]) to one. To do so, it sets the value of this variable in each position to the value of the pbl in the range of the area of the packet.

(9). This last constraint specifies that if two packets are being used in the same truck, the cells occupied by one packet cannot be occupied by another one. In plain words, this constraint makes sure there is no overlapping in all trucks.

4. Metaheuristics

In this project two different approaches for metaheuristics have been used: Greedy Randomized Adaptive Search (GRASP) and Biased Random-Key Genetic Algorithm (BRKGA). In this section the basic structure of the algorithms is introduced in pseudo pythonic code.

GRASP

Grasp is iterative process that consists in two phases, a construction phase, in which a feasible solution is produced, and a local search phase, in which a better solution is created by searching the local optimum in the neighborhood of the first constructed solution. Finally, the best overall solution is kept as the result.

```
Def grasp(config, problem):
      bestSoltion = ∅
      while(iterations, time or quality):
            solution = constructivePhase()
            solution = localSearch(solution)
            if (solution.load < bestSolution.load):
                  bestSolution = solution
```

In the case of this problem the constructive phase is a pseudo greedy approach that constructs a solution one element at a time. First, all the possible candidates for a given element are retrieved and sorted by the load they add to the solution in ascendant order. Then, the list is restricted to a percentage (alpha) of the best available candidates, this list is called *restricted candidate list* (RCL). Finally a random candidate is selected from the new constructed RCL and added to the solution. This technique allows to create different solutions in each iteration.

```
Def constructivePhase():
      Solution = ∅
      For package in allPacakges:
            candidateList = findCandidates(package)
            sortedCL = sort(candidateList) #by load in ascendent order
            minLoad = sortedCL[0]
            maxLoad = sortedCL[len(sortedCL) – 1]
            boundaryLoad = minLoad + (maxLoad - minLoad) * α
            maxIndex = 0
            for assignamentPackage-CPU in sortedCL:
                  if (assignament.Load < boundaryLoad) ++index
            RCL = sortedCL[0:maxIndex]
            Solution.append(random(RCL))
```

```
def FindCandidates(package):
      candidates = ∅
      for truck in allTrucks:
            feasible = False
            for all positions in truck.layout:
                  if (position is free):
                        feasible = True
                        for all positions in package.area
                              if (position occupied):
                                    feasible = False
            if (feasible):
                  load = getLoad()
                  candidates.append([package,truck,Load])
      return candidates

def getLoad():
      load = usedTrucks * truckWeight
      weight = 0
      for all trucks in usedTrucks:
            wTruck = 0
            for all package inside truck:
                  wTruck += package.weight
            weight = max(weight,wTruck)
      load += weight
      return(load)
```

As can be seen in the pseudocode one truck is candidate to be assigned to a packet only if it has the necessary available space for that package. To check whether package fits a truck, a double matrix search must be performed. Each truck stores a matrix representing the available and assigned space in the truck. Initially a free position is search in the layout, this position becomes a candidate to store the package, at this stage we can assure the package fits so we perform a second search from that candidate position to the whole area the package would occupy, if we find a single occupied space in the search a new candidate position must be found.

If the truck is found to have space for the package, a candidate is created. A candidate is formed by the package to store, the truck in which it is stored and the load the candidate adds to the on creation solution. To calculate the load we store the amount of trucks being used until the moment and this value is multiplied by the weight of the truck. The load also includes the overall weight of the max loaded truck.

Local Search

The previous construction is not guaranteed to be a local optimum, thus, is a good idea to launch some kind of local search in order to find the best possible neighbor in the solution local space. Two options are implemented for the local search, a reassignment in which iteratively each package is tried to get assigned to other truck. Exchange is the second option, in this approach a package is reassigned to the truck of another assignment, and the same happens in the reverse, this is done until all assignments are exchanged.

```
Def localSearch(solution):
      OptimizedSolution = solution
      assignaments = solution.getAssignaments()
      If (reassignment):
            For assignament in assignaments:
                  Package = assignament.package
                  prevTruck = assignament.truck
                  For newTruck in trucks:
                        neighbor = solution
                        neighbor.deAssing(package, prevTruck)
                        neighbor.assign(package,newTruck)
                        if(neighbor.load < optimizedSolution.load):
                              optimizedSolution = neighbor
      else if (exchange):
            for i in len(assignaments):
                  package1 = assignaments[i].package
                  truck1 = assignaments[i].truck
                  for j descending from len(assignaments) to i:
                        package2 = assignaments[j].package
                        truck2 = assignaments[j].truck
                        neighbor = solution
                        neighbor.deasing(package1,truck1)
                        neighbor.deasing(package2,truck2)
                        neighbor.assign(package1,truck2)
                        neighbor.assign(package2,truck1)
                        if(neighbor.load < optimizedSolution.load):
                              optimizedSolution = neighbor

      return OptimizedSolution
```

BRKGA

BRKGA is an example of genetic algorithm, which apply the concept of evolution to find optimal or near-optimal solutions. One solutions is an individual in a population which has a chromosome that codes the solution. Chromosomes are a list of genes that can take values from 0 to 1. At the same time, a chromosome has a fitness level which is linked to the objective function of the problem. Individuals are then evolved to create a new population, though some of the individuals that have the highest fitness of the whole population are maintained. The evolution of individuals is performed in two different ways: one possibility is to completely mutate all the genes of an individual, generating completely random values, the other solution is to create a crossover between the elite individuals and the non-elite, making sure there is more change of gene inheritance from the elite than from the non-elite.

This approach generates an algorithm with a problem independent part, which is the evolution of the population, and a dependent one which is the correlation between the chromosome and the actual solution. This dependent part is called *decoder* and it is the only thing the user has to code.

Solution encoding

Each chromosome contains as genes as the number of packages $nGenes = |\ nPackages\ |$ such as the $i - th$ gene corresponds to the $i - th$ package. The genes in the chromosome codes the probability of a package to be assigned earlier in the execution.

Chromosome decoder

Packages are joined in tuples with its corresponding gene, the area of the package alongside the probability coded in the gene determines the position in which the package is going to be assigned. A higher gene value alongside a huge area assures the package is going to be assigned early.

```
Def Decoder(Individual):
      #create a list with tuples containing a package and a gene with
      the structure [(p1,g1),(p2,g2),…,(pnPackages,gnPacakges)
      Packages_and_genes = zip(packages,genes)
      #sort in descendent order
      Packages_and_genes.sort(sort by package.area * genValue)
      Individual.solution = ∅
      For all tuples in package_and_gene:
            Package = tuple.package


            #To find the best assignament for a package we follow the
            same code as for the construction of the candidate list in
            GRASP, though in this case we return only that candidate
            truck with the lowest load
            truck = findAssignament(Package)
            individual.solution.assign(package,truck)


      #When all packages are assigned to a truck the fitness of the
      individual is calculated. In this case the fitness is the value
      of the optimization function for this solution. We recover this
      value with the same code we did in GRASP


      Individual.fitness = individual.solution.getLoad()
```

5. Linear programming versus Heuristics

In this section we are going to compare the results obtained when launching problems to IBM's OPL against the proposed metaheuristics algorithms.

Problems

The problems executed in this test are all manually crafted seeking to search the behavior both OPL and metaheuristics under certain conditions. The problems start with extremely easy values and keep growing not only the input data, but also, the complexity.

- **Problem 1**: A really easy execution, it contains a small truck and a small packet.
- **Problem 2**: An evolution of the previous problem, it contains the same truck but with a packet that fits the truck.
- **Problem 3**: A single truck and four different packets that fills all the space.
- **Problem 4**: A single truck and six different packets filling the space. The packets are crafted so there are only one or two possible combinations.
- **Problem 5**: Duplicate the previous problem. We increment the available truck weight to 1000 so it is easy to check if the optimization value is correct.
- **Problem 6**: Increment one truck and six more packets to the previous problem. The dimensions of the packet make it infeasible to create many changes in the layout of the truck increasing the complexity of the problem. All trucks must be 100% occupied with little possible changes in the packet position.
- **Problem 7**: increment one more truck but not the number of packages. It is expected to see the same optimization value than before but one less used truck.
- **Problem 8:** Increment in six the number of packages. Now as in previous executions all trucks have to be filled.
- **Problem 9**: New problem from scratch. More real data, seven trucks and 23 packages with small areas, there are some small incompatibilities between the packages.
- **Problem 10**: Evolution from the previous problem, now with higher truck space that should add more complexity.
- **Problem 11**: added even more truck space to the previous problem.

Linear Programing results

OPL gives surprising results to some of the problems.

Problems 1 to 4

The small dataset used in this executions causes OPL to find the optimum solution in less than a second even when adding problem complexity in the 4th problem that allows minimum changes in the truck layout.

Problem 5 to 7

Problem 5 starts to get really interesting, in this problem we can see an increase to almost 4 seconds of computation by only adding one more truck and a few packages to the problem. Problem 6 on the other hand increases even further the computational time to a minute by adding one more package. Finally problem 7 adds a useless truck to the mix, the objective was to see if the minimization would leave this last truck unused and with an optimum value equal to the problem 6. At the same time the computational time is increased a few more minutes because of the new tuck.

These problems demonstrates that the number of trucks as well as the layout they have, are two of the most demanding values in the problem.

Problem 8

Initially we expected to see a behavior similar to the evolutions of problem 5 6 and 7, getting a computational time of approximately 10 minutes, the results though are quite different. After four hours of calculation a solution appears with a correct optimum value. This is an example that minimum changes and high complex problems can make OPL behave exponentially.

Problem 9

This problems features more input data than the other problems, high number of trucks and packages, but resulting in lower execution time, the conclusion is that even higher input data can result in low calculation time if the problem is easy.

Problem 10

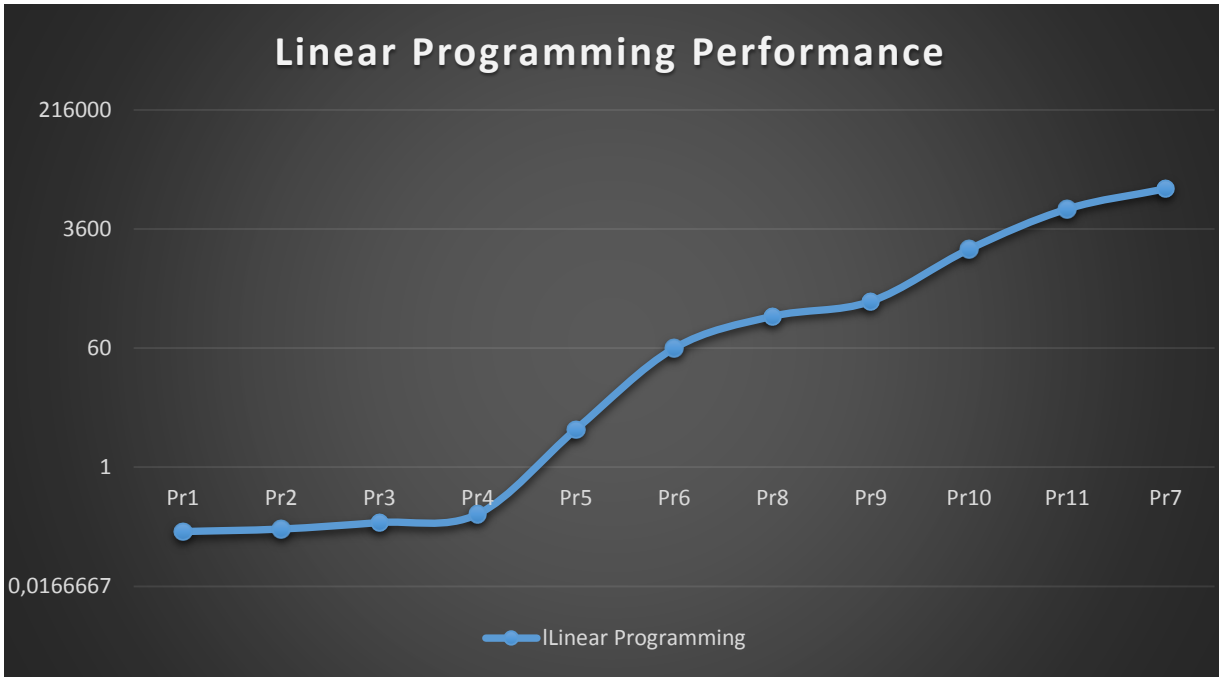
The added truck space increases dramatically the execution time to more than 10 minutes.

Problem 11

Even though this problem only has one more truck than the previous problem the computational time increases to almost two hours.

Overall performance

All the results are gathered and sorted by their execution time resulting in a graph that shows the execution time from easier problems to complex ones. It is easy to see that OPL is really fast with small and easy problems but it keeps increasing the execution time as the size of the input data increases and so does the complexity. The behavior of Linear Programming is to reach exponential computational times when the problem grows high in input data or complexity of the solution.



The graph shows the performance of Linear programming under eleven workloads, the left axe shows the time in seconds in logarithmic scale for the completion of the given task. As it can be seen, Linear programming behaves exponentially, as the complexity of the task or the input data increases, so it does the time required to solve a problem.

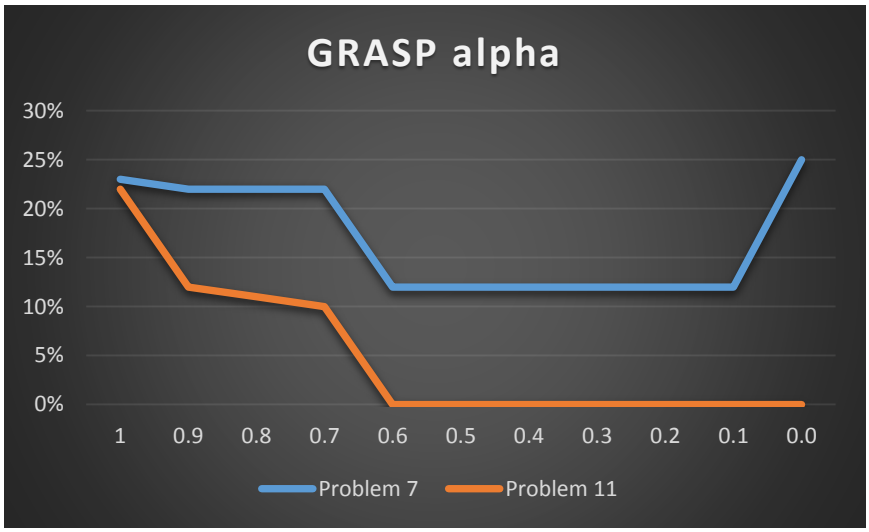
Metaheuristics results

Heuristic algorithms appear to solve the limitations approaches as Linear Programming shows when the problem grows in complexity. For really big problems Heuristic solutions shows a tradeoff between execution time and quality of the solution; many times the solution proposed by these algorithms is not the global optimum for the problem but offers the solution in an acceptable times.

Both BRKGA and GRASP solutions introduced in these document have special parameters to tune in order to get the best possible performance, so, the first action to be done is calculate the best possible values for Alpha (GRASP) and the number of individuals (BRKGA)

α Value for GRASP

In order to get the best possible alpha we have taken two of the biggest problems to analyze, deactivate local search and see the optimization value shown in a single iteration. The results of the tests are the following:

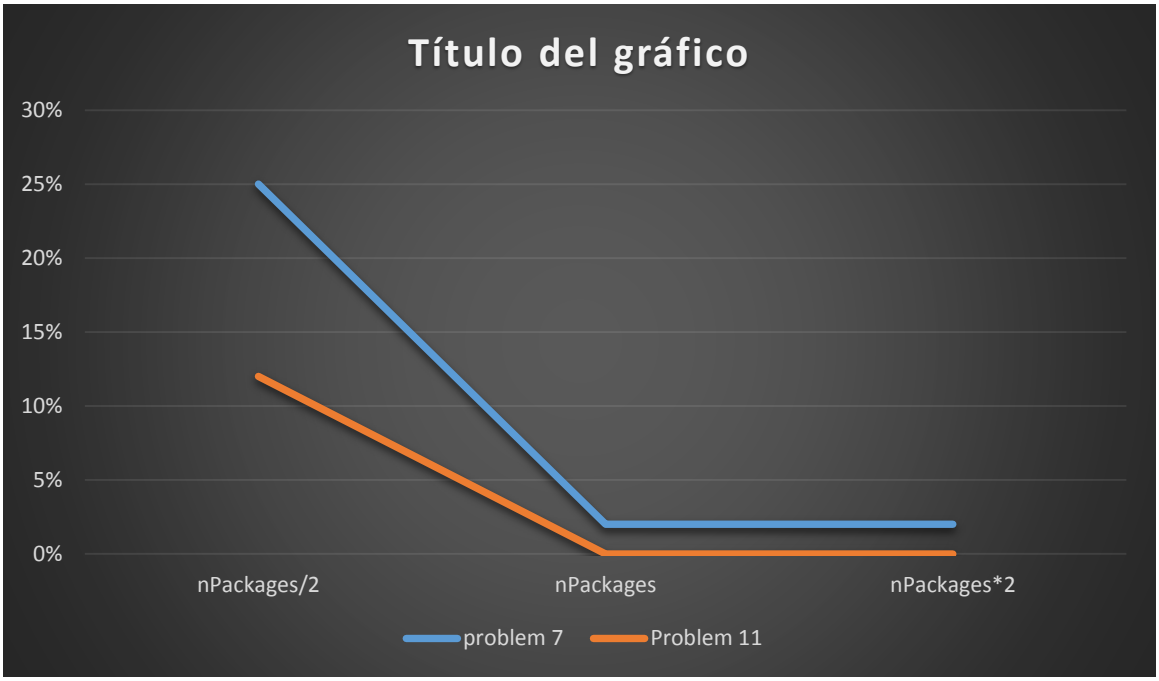


The graph above shows the percentage difference between the value of the first solution of grasp regarding the value of alpha and the real optimum solution. We can see a clear progression of improvement when the alpha reaches the values from 0.6 and 0.1, values that gives a fair amount of randomness to the construction. The first problem shows that GRASP can hardly find a good constructive solution and needs local search to improve the solution though moving the alpha to values between 0.6 and 0.1 exceptionally improves the first solution.

The second problem shows high performance in the constructive phase for all values of alpha, giving the optimum directly in the constructive phase with alpha values lower than 0.6. As we see a drop in performance in the first problem when reaching values of 0.1 and 0, the selected alpha value is going to be 0.4.

Number of individuals for BRKGA

As in the previous test we are going to perform a single iteration of BRKGA with three different values of the number of individuals: $nIndividuals = |nPackages|$, $nIndividuals = |\frac{nPackages}{2}|$ and $nIndividuals = |2 * nPackages|$. As before problems 7 and 11 are the candidates for the tests.



BRKGA shows a similar behavior regardless of the problem executed, setting the number of individuals as the half of the number of packages results in an initial solution far from the optimal. As we increase the number of individuals on the population, the results of the first iteration start to get closer to the wanted optimum. Though we could keep increasing the number of individuals to get better and better results, the performance in computational time decreases accordingly. Due to this we have settled the number of individual as the number of packages.

Problems 1 to 6

As we would expect all the heuristic options work really fast, getting the optimal solution mostly at the first iteration, in so small instances local search seems to make GRASP a little bit slower.

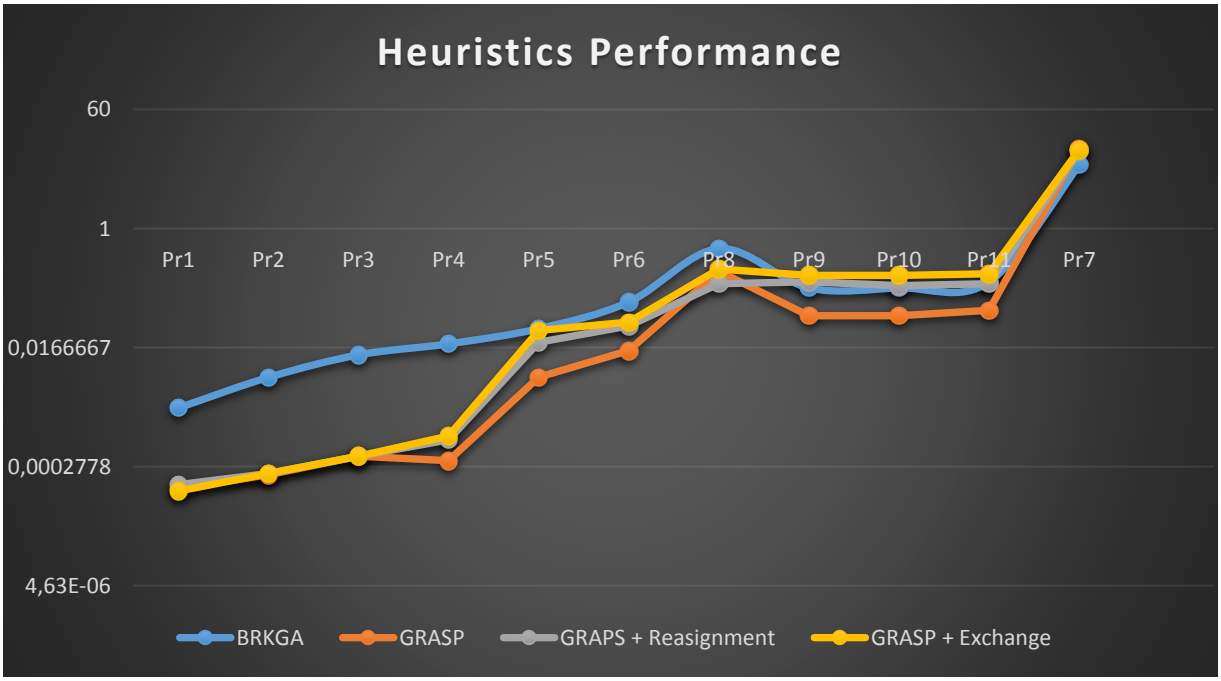
Problems 8 to 11

In problem number eight all heuristic algorithms tend to noticeable increase the computation time, this is an expected result as the input data has grown significantly from the last problem. The surprise arrives when executing problem number nine, it was expected to perform worse than the previous problem, though the reality is that the performance is increased. This behavior can also be seen in the next problems, the amount of input data is heavily increased though the performance remains mostly the same. This demonstrates that heuristic problems are not as affected to input data as OPL.

Problem 7

The last problem in our set shows that increasing the complexity of the problem harms the performance quite a lot. Though all algorithms are capable of reporting a solution fairly soon, reaching the global optimum takes a lot of time.

Overall performance



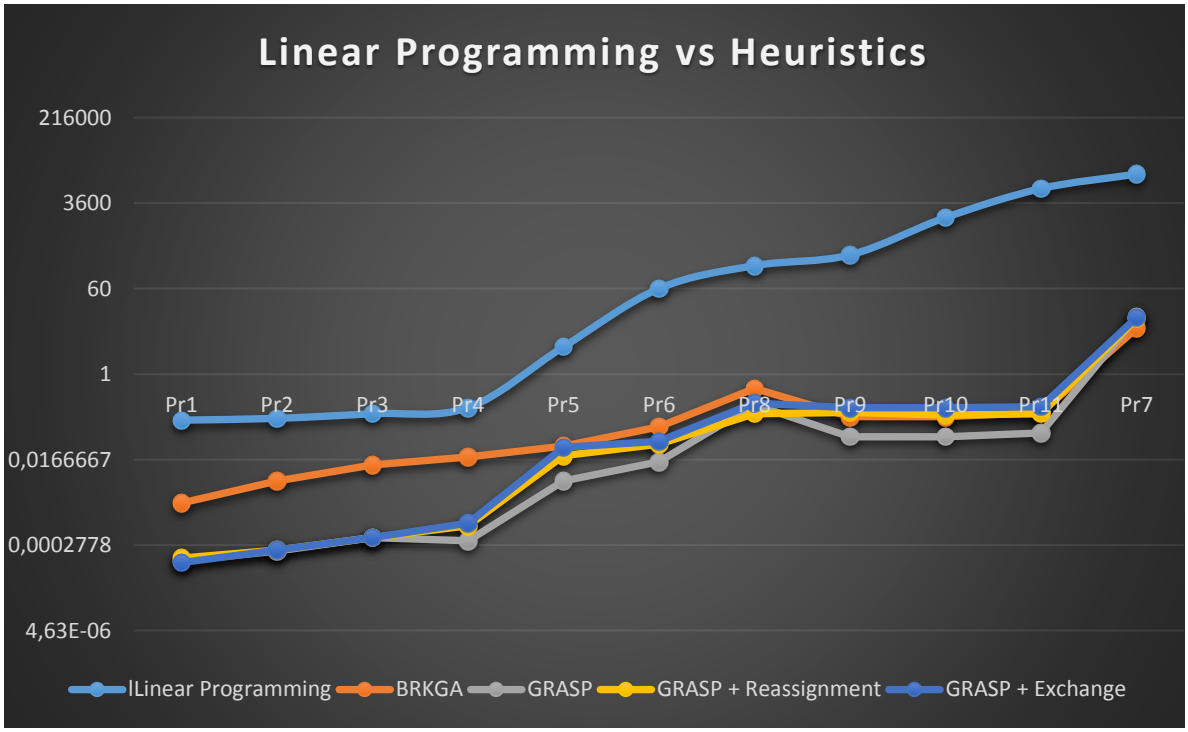
The graph above shows the performance of BRKGA and GRASP with and without local search. It can be seen than most of the problems are solved in milliseconds, a remarkable compared with the performance of OPL. A few conclusions can be extracted from this results, first of all, for small problems the overall performance of the four approaches is quite similar, though for really small problems BRKGA seems to carry a little bit more of overhead.

The second conclusion is that local search mostly harms GRASP performance, this is most probably due to the small problems we are trying to solve where searching for neighborhoods may not be necessary to increase the quality of an already good constructed solution.

Last conclusion is that, at least for small problems, heuristics does not seem to be really affected by input data as it does by problem complexity. This means this kind of approaches are designed for big problems that cannot be solved with any other tool.

Overall performance comparison

The next graph shows the performance comparison between Linear programing, BRKGA, GRASP, GRASP + reassignment local search and GRASP + exchange local search. The Y axis shows the time to compute one problem, the axis is set in logarithmic scale to avoid most of the executions to be near cero.



Once again a few useful conclusions can be extracted from this analysis: First of all Linear programming will always give you the optimal solution for a given problem, the cost is a low performance in time of computation.

Linear programming is heavily affected for both input data and problem complexity, in our case, try to solve a problem that has several trucks with many packets that can only fit in specific positions may lead to hours of computation. But, at the same time increasing the problem size but not its complexity will show that behavior as well, for example, if we add a few more trucks but not more packages, in a logical perspective our optimal solution will be exactly the same as in the previous problem and excluding the added trucks. OPL though, checks every single possible solution creating a really hard to solve problem that may last for hours.

Heuristic algorithms on the other hand show an incredible boost in performance compared to OPL, all executions are performed in milliseconds or in a few seconds compared to the hours of computation OPL needs to find the optimal solution. This gives a clue that metaheuristic solutions are a great option for combinatorial problems such as the one presented in this report.

Even though heuristic approaches seem to not be affected by the amount of input data the problem has, they are heavily affected by problem complexity. Really big problems with tenths of trucks and hundreds of packets may be solved fast but a relatively small problem with a complex solution may make even metaheuristics struggle.

All in all this comparison is useful to see the differences in performance of two of the most famous solvers for combinatorial problems but the real performance of metaheuristics cannot be evaluated with such as small problems as the ones used in OPL, so, a new comparison is needed, this time with really big and complex problems.

6. Metaheuristics performance comparison

As stated in the previous section, to properly evaluate the performance of heuristic approaches, big and complex problems are needed, to create this problems an instance generator has been used.

Instance generator

```
Def InstanceGenerator(difficulty, nTrucks):
      nTrucks = nTrucks
      nPackages = random(nTrucks *multiplier, nTrucks *multiplier)
      xTruck = #Set by the user
      yTruck = #Set by the user
      wTruck = #Set by the user
      for all packages:
            xDimp.append(random (#userSpecified,#userSpecified))
            xDimp.append(random (#userSpecified,#userSpecified))

      for 0 to random(0,5):
            incomp[random(0,nPacakges)][random(0,nPacakges)] = 1
      #all this values are added to hashmap problem with the structure
      problem["nTrucks"], Problem["xDimp"]…
      Def check(solution, difficulty):
            areaPackages = ∑_{p=0}^{nPacakges} areaPacakge
            areaOccupied = areaPacakges / (xTruck*yTruck*nTrucks)
            #The difficulty level specifies the total area occupied by
            the packages inside the truck
            if (areaOccupied >= difficulty):
                  weight = ∑_{p=0}^{nPacakges} pacakge.weight
                  if (weight <= wTruck * nTrucks):
                        saveSoltuionInFolder()
                  #The instance is saved in the data folder of the
                  execution framework
```

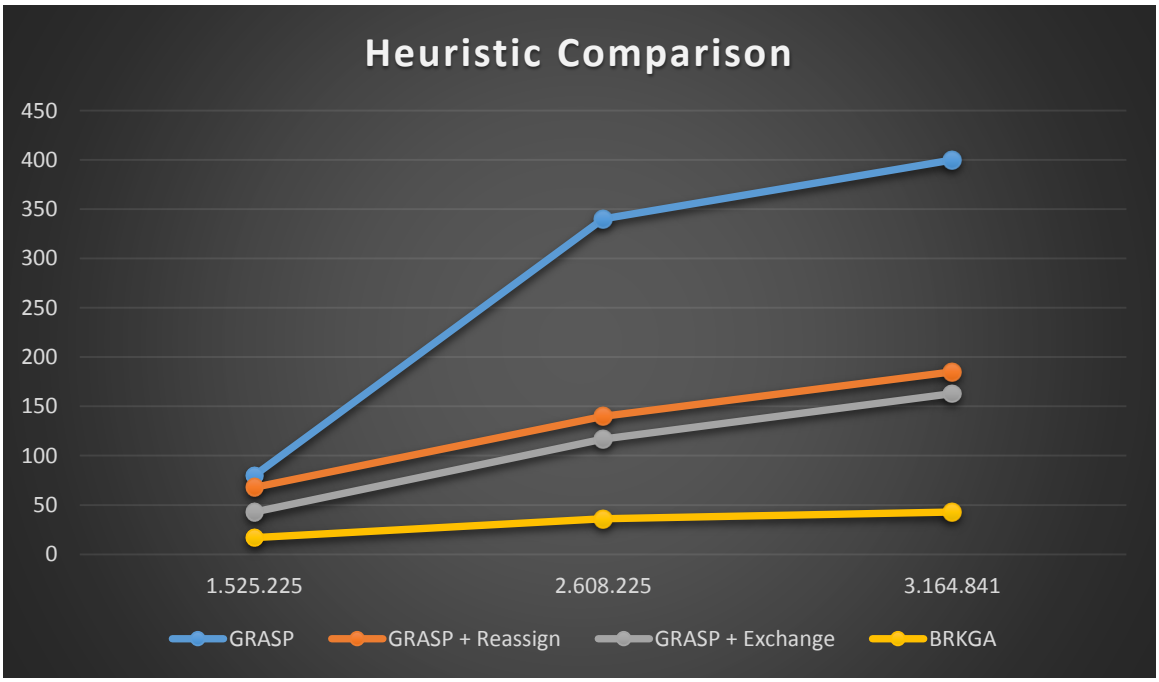
Execution specifics

As we cannot guarantee the solution provided by the algorithms is the local optimum, we keep iterating until there has been 10 minutes since the algorithm has found the las best solution. In addition as the best tuning parameters have been already been discovered in the comparison against Linear programming the same values are used the following executions.

The first execution of the test is performed to locate the best solution all the algorithms can calculate, this solution is our new objective for the next iterations. The objective of the test is calculate how much time it takes for each algorithm to reach for the previous calculated best solution.

The size of the problem is defined by the difficulty problem multiplied by the area of the packets and the area of the trucks. We have defined the difficulty of the problem as the total area occupied of the packets respect to total space of the trucks.

Comparison



The graph above shows, for the Y axis, the seconds an algorithms has spent until it finds the best solution while the X axis represents the size of the problem. The graph shows interesting results, on one hand the obvious best contestant is BRKGA, with huge differences of performance compared with its counterparts. BRKGA seems to respond linearly to increased problem size.

Both local search options for GRASP have surprisingly good performance, the two of them are pretty similar though most of the times Exchange seem to perform a little bit better. The reason is that it exchanges all possible assignments for each solution in the search for optimizations while Reassignment changes the truck assigned to each packet, aiming to improve the quality of the solution.

Finally, as logic would tell us, GRASP is the worst contestant in the rig, the performance is far worse than any possible local search optimization or genetic algorithm and tends to perform even worse as the problem size increases.

The main conclusion of this experiment is that heuristic approaches are a great way to search for optimal or near-optimal solutions for huge combinatorial problems, a greedy function may be enough in lots of cases but in those problems with millions of possible combinations the performance decrease noticeable. In those cases one of the best possible approaches is to use a genetic algorithm or a pseudo greedy construction with random behavior and a local search algorithm to tweak and improve the solution.