Stochastic Generalized Bin Packing Problem

Cecilia Bartoletti, Valeria Bo, Federico Sisci

February 17^{th} 2023

Contents

1 Description of the problem

The problem considered is the "Stochastic generalized bin packing problem". There is a set of items to be shipped, each one with a specific volume and profit, using a set of bins, each one with known capacity and cost.
Items are divided in compulsory and non compulsory while bins can be of different types; bins of the same type are characterized by the same capacity and same cost. A maximum number of bins of each type is fixed.
Our goal is to place all the compulsory items and as much non-compulsory items we can in our bins in order to minimize the total cost given by the difference between the costs of the used bins and the profit of the loaded non-compulsory items.
Now we define some notations we are going to use:

- a set J of bins;
- a set I of items divided in I^C compulsory items and I^{NC} non-compulsory items;
- C_j the cost of each bin;
- p_i the profit of item i;
- w_i the weight of item i;
- W_j the capacity of bin j;
- B the maximum budget available.

The problem can be expressed as:

$$min \sum_{j \in J} C_j y_j - \sum_{i \in I^{NC}} p_i \sum_{j \in J} x_{ij} \tag{1}$$

$$s.t. \sum_{i \in I} w_i x_{ij} \leq W_j y_j, \qquad \forall j \in J \tag{2}$$

$$\sum_{j \in I} x_{ij} = 1, \qquad \forall i \in I^C \tag{3}$$

$$\sum_{j \in I} x_{ij} \leq 1, \qquad \forall i \in I^{NC} \tag{4}$$

$$\sum_{j \in J} C_j y_j \leq B \tag{5}$$

$$x_{ij}, y_j \in 0, 1, \forall i \in I, \forall j \in J \tag{6}$$

(1) is the objective function that minimizes the total cost where the total profit of the compulsory items is not considered since it's a constant. The constraint (2) requires not to exceed the bins' capacities, while (3) force the selection of all the compulsory items exactly one time and (4) force us to take all the non compulsory items no more than one time. The last constraint (5) is about the maximum budget available.

2 Instances

In *istances.py* we have defined the parameters of our problem. Two possible sets of values are introduced and they differ from each other by the profit and the weight values of the items. In both cases the number of items is set to 100 because the academic's Gurobi can't perform with more data.
Moreover, we have decided to fix the number of bins to 5, each one with a capacity randomly selected from a uniform distribution that takes value between 20 and 40. Similarly each bin's cost was randomly selected from $U \sim [100, 120]$. The maximum budget was set to 400 in order to make the Heuristics use as less bins as possible.
Let's now introduce the difference between the two sets of parameters:

- in the first one, both the profit and the weight of each item are randomly selected from a uniform distribution ($U \sim [10, 12]$ and $U \sim [1, 5]$ respectively)

- in the second one each item's profit comes from a Poisson distribution with parameter equal to 8, so it takes value approximately between 0 and 16 with an higher probability in the neighbourhood of 8 (its mean value). As far as the weight of each item concerns, we chose to randomly select them from a Gamma distribution with parameters $a = 2.5, b = 1$, where its mean value equals the one from $U \sim [1, 5]$.
  We chose two distributions that assume only positive values, since the variables they represent cannot be negative.

Finally, the set of parameters, we are going to use with the different heuristic methods, is selected randomly at the beginning of the script through a variable called index $\in \{0, 1\}$: if index is equal to 0 the first set of values is used otherwise the second one is selected.

3 Gurobi

The first method used to optimize the problem required the academic Gurobi software. Gurobi is a solver identified by the python package *gurobipy*.
After installing the package, we defined the model to optimize through the command $grb.Model('gbpp')$ and we proceeded setting the two binary variables Y for the bins and X for the items:

- $Y = 1$ if bin j is rented
- $X = 1$ if item i is collected by bin j

Then, we defined the objective function of our minimization problem and we add it to the model.
Through $model.addConstr$ we established all the necessary constraints characterizing our problem introduced in the first chapter: every compulsory item has to be selected, the non compulsory items can be taken at most once and the constraints regarding the capacity and the budget.
As far as the stopping criterion concerns we set two variables:

- $MIPgap$: difference between two consequential solutions. If we reach that value the process stops
- $TimeLimit$: in our case after 360s Gurobi is stopped.

Then, after we have created the model, we save the results in a $Logfile$ and we optimize the model using the command $model.optimize()$. Finally, we print which set of instances we have used and using a for cycle we print, for each bin, the bin used and the compulsory and non compulsory items contained.

4 Customized Heuristics

To solve this problem we have decided to use the Next Fit Algorithm in order to create an equal initial solution for both the customized heuristics that wasn't already an optimal one and that had a low computational cost.
In the first customized heuristics we have decided to create a Tabu Search algorithm that took in input the solution of the Next Fit and the instances.
The second one was the Genetic Algorithm that, as the previous one, takes in input the same Next Fit solution and instances.

4.1 Next Fit

The Next Fit is an approximation algorithm that in every instant has only one open bin.
This means that we take all the items one by one in order and we allocate them, if it's possible, in the open bin and if is not, because there isn't enough space, we open the next bin and try to allocate the selected item in there.
The Next Fit function used takes as input some of the instances and provides as output:

- solution: a binary matrix of the solution where the number of row is the number of items and the columns are the bins.

$$\begin{cases} solution(i,j) = 1 & \text{if item } i \text{ is in bin } j \\ solution(i,j) = 0 & \text{otherwise} \end{cases}$$

- bins: a binary vector equal to 1 if the bin is taken and 0 if is empty
- residual_capacity_bin: a vector with the residual capacity of each bin
- allocation: a vector where each item has the bin where it is placed. If it isn't in any bins $allocation(i) = 0$

Inside this function we first ran the main Next Fit's algorithm placing the items in the *allocation* vector, then we initialize the *bins* vector with the *np.ones* command because we are taking all the bins and in the end we create the *solution* matrix and the *residual_capacity_bin* vector.

4.2 Tabu search

The Tabu search is a Local Search Algorithm but the difference between this method and a classic Local Search Algorithm is that the Tabu Search can memorize some information about the last solutions making possible to get out from local best solution.
The implemented algorithm is however a simplified Tabu Search because most of the function we have created didn't use the Tabu Lists to have memory of the last moves because these functions examine the neighborhood, saving the solution when the objective function gets better and then go along without the possibility of doing again the previous action.
For this reason a Tabu List would be useless.
For this algorithm we have created a file called *Tabu_Search.py* that contains the function *tabu_search*.

First, inside the *tabu_search* function, we have implemented 3 action to find the local best solution:

- *put out*: is a function that substitutes an item inside one of the bin with a "left out" item in order to minimize the objective function
- *swaps*: is a function that "swap" two items in two different bins in order to create more space to add new items inside one of the bin
- *eliminate*: is a function that inside has other 3 functions with the purpose of eliminating the bins with a cost higher than the profit of the items inside

After these functions, we applied the function *all_combination* in order to try to exit from the local best solution and finally the *budget_constrain* function to maintain the constraint about the maximum budget respected. All this program will be ran twice.

4.2.1 put_out function

This function takes as input the best solution matrix, the item that we want to add to the bin, the item that we are removing from that bin and the bin. The output is the new best solution matrix.
In order to use this function we have created a for cycle that takes all the non compulsory items inside the bins and all the non compulsory items outside the bins and if the taken left out item has an higher profit than the one inside and a weight lower than the residual capacity of the bin plus the weight of the item inside then we apply the function and update the cost, the best solution and the residual capacity.

4.2.2 swap function

This function takes as input the best solution matrix, the item (compulsory or not), the bin where the item is, an other item and its relative bin. We *swap* the position of the two items inside the two bins and the output is the updated best solution matrix. In order to use this function we have created a for cycle where we take the two items and the relatives bins and then we take the heaviest one and we check if the weight of the other item plus the residual capacity of the relative bin is big enough to contain the first item. If it is we apply the function, update the residual capacity and make a for cycle trying to add an item from outside in the bin where the heaviest item was before.

4.2.3 best_option function

This is one of the function inside the eliminate function.
The input are a list of the non compulsory items inside a taken bin, a list of the relatives profit, a list of the relatives weight, one compulsory item and the residual capacity of the taken bin.
Firstly this function orders the list of non compulsory items from the one with lower profit to the highest and it creates a list of the first n items of the ordered list that are needed to make up for the lack of capacity of the active bin to host the compulsory item. Secondly it does the same but with a list of the ordered items from the one with higher weight to the lowest.
At the end of the function we compare the two values corresponding to the loss of profit in the two cases and we take the option with a lower value.
The output is the value of the lost profit, a list with the items that we have to take out from the bin in order to make space for the compulsory item and the number of items inside this list.

4.2.4 eliminate function

This function takes as input the best solution matrix, the vector of the residual capacity of the bins, the binary vector that signal if the bins is used or not and the bin we want to empty.
First of all we use the function *erase_bin* that save in a binary vector "add" all the compulsory items inside the bin that we want to empty.
After we apply the function *swap_out* that take as input the temporary solution obtained from *erase_bin*, the "add" vector, the vector with the residual capacity, the bin we want to empty and the binary vector of the bins.
Inside *swap_out* we make for each compulsory item with value 1 inside the "add" vector a for cycle over the still active bins, different from the one we want to empty, then we create a list of the non compulsory items inside this active bin checking that the list is not empty. If it isn't empty and the residual capacity of the active bin plus the sum of the weight of all the non compulsory items inside is high enough to contain the compulsory item we apply the function *best_option* and we save all his output inside three lists and add to another list the bin used in this iteration.
Then we check that there is at least one bin able to hold this compulsory items, if there isn't we stop the function returning an error.
If there isn't an error we find the combination of items that give the lower loss of profit and we update the solution matrix, the residual capacity vector and the "add" vector.

After we have applied the *swap_out* function we have to check if an error occur. If this is the case we can't empty the bin because there is at least one compulsory item left out and because of the relatives constraint this can't happen.
If the error didn't occur we can update the bins vector, the residual capacity vector and the best solution matrix.
In order to use this function we have made a for cycle over the active bin where we try to empty it if the profit of the bin is strictly positive.
If the elimination of the bin can be done we update the results otherwise we leave the bin active and we keep going with the for cycle.

4.2.5 all_combination function

This function takes as input the best solution matrix and the quantity of capacity we have to fill. Inside this function firstly we create a list with all the non compulsory items that haven't been taken, order them from the one with lowest weight to the highest one and then we select only the items with a weight lower than the one we have provided in the input.
Secondly, with a for and a while cycle we create all the possible combination starting with the selected item and in the end we select the option with the highest profit.
The outputs of this function are a list of the combination with highest profit and the relative profit value.
In order to use this function we have created a for cycle that for every bin create a list with the non compulsory items inside that bin and another list with the relatives weight. Then we order the items from the one with highest weight to the lowest one and in another for cycle we use the *all_combination* function to try to substitute every selected item.
Finally if the profit passed as output is higher than the taken item's profit we substitute the selected item with the items in the output's list otherwise we don't change anything and go on with the for cycle.

4.2.6 budget_constrain

This function has the scope to find the best bin to empty if the budget constraint isn't satisfied.
The input are the value of the max budget, the budget we are currently using, the best solution matrix, the residual capacity vector and the bins vector.

Inside this function firstly we create a list with the still active bins and a list with their contribution to the total costs. Then we order the list of bins from the one one with highest contribution to the lowest one because we want to try to free first the bins that would cause a lower rise of the objective function.
Then, if there is at least one bin with an enough high cost to achieve the purpose, we try to empty these bins. If there aren't bins able to do such thing or if these bins can't be freed we try to find the combination of bins able to obtain a lower budget thanks to a while cycle.
The outputs are the vector of bins, the best solution matrix, the residual capacity vector and the error value.
To use this function we first have to ask if the budget constraint is respected and if is not we apply the *budget_constraint*.
If the error passed as output is equal to 1 it means that is impossible to empty enough bins in order to have all the constraint satisfied, whilst if the error is equal to 0 we can update the data with the function's output.
Finally if the budget constraint is respected and there is enough space for one of the empty bin we can try to fill it with the *all_combination* function, passing as second input the total capacity of the bin. If its contribution help to minimize the objective function we add it and update the data with the new solution.

4.3 Genetic Algorithm

The Genetic Algorithm (GA) is a customized heuristics inspired by the process of natural selection.
In general, GA uses three functions (*mutation*, *crossover*, *selection*) taking example from nature. His structure is:

- Fix an initial population of m solutions;
- Generate a new set of k solutions using the functions of *mutation* and *crossover*;
- Apply *selection* to find the m most fit solutions to be used at the next iteration;
- Stop the algorithm after a fixed number of iterations or when the m solutions found are enough close to each other.

In our problem, we have decided to modify a bit this sequence of steps creating more function to better adjust the algorithm to our problem.

First of all, we have created a file called *Genetic_Algorithm.py* which contains the function *genetic_algorithm* and all the additional functions used in it, that are:

- $is\_feasible$ to check all the constraints in our minimization problem
- *change_random_gene* used in the *mutation* function to change a component of a specific element of the population
- *mutate* to change the bin selected before with another (or no one) for a specific item
- *pick_two_parents* to select two specific solutions of the population for the *crossover* function
- *crossover* to crossover the two solutions selected in a random point and to create a new solution
- $objective\_function$ to compute the value of objective function and a vector of bins used or not of our minimization problem
- *bin_mutation* to start with an initial solution that already satisfies the budget constraint

We start the algorithm copying some elements passed as input in the *genetic_algorithm* function and we apply the *bin_mutation* function to the initial solution, *allocation* , passed from the $next\_fit$ algorithm called here pop_nf. Then we create three vectors: *population*, $value\_obj\_fun$ and *min_bin* to collect respectively the solution, the value of the objective function associated to it and computed by the $objective\_function$ and the vector of bins used in this solution.
The initial population is taken from the *bin_mutation* function.
Then a while cycle begins and it doesn't stop until the dimension of the population is smaller than the parameter *size_population* defined in the *main.py*. At each iteration we do the mutation, append the new solution to the population vector, compute the value of objective function and append it to $value\_obj\_fun$ and the same thing we do with min_bin. Then we do the same with the crossover operation and we save the smaller of all the values we have tested and the minimum value saved is the solution of our minimization problem. In particular we save also the position of the minimum value because, in the same position but in the vector *min_bin* there's the vector of bins used associated to the best solution.
This procedure is the same at each iterations except for the first one because

the crossover needs more than two elements to work good, for this reason we do only the mutation in the first iteration.
Now let's discuss the functions in GA.

4.3.1 bin_mutation

This function modifies the solution provided by the Next Fit algorithm into another one that respects the budget constraint. First of all we create two lists *bin_el*, where we'll save the bins deleted in order to reach our goal for the budget and *add* that contains the compulsory items to replace in the other bins. Next we compute the budget used by the current solution saving it in the variable *bg*.
Then we introduce a *while* cycle: as long as *bg* is greater than our maximum budget, a random bin is selected and we insert it in the list *bin_el* if not already in it. Every time a bin is added in *bin_el* we update the variable *bg* subtracting from it the cost of bin c. Finally, we check if the bins eliminated contained one or more compulsory items, if so, we save these items in the list *add*; moreover, we remove any non compulsory items found in the bin considered.
This list will be used in the algorithm to reinsert the compulsory items, that need to be reallocated thanks to the *bin_mutation* function, in the solution vector.
Then we update the solution vector and we save it as first element of our population.

4.3.2 is_feasible

This function checks if the population after mutation or crossover is feasible, this means that all the constraints in the minimization problem have to be satisfied. The input is the new solution and the output is a parameter $is\_feasible\_solution$ equal to one if the new solution is unfeasible and zero otherwise.
First of all we check if the length of new solution is the same as the number of items; if it's true we proceed with the others constraints.
Then we check if, for each bin, the sum of the items' weights in a particular bin is bigger than the capacity of the bin considered. To do this we create a vector as long as the number of bins where the i position in the vector represents the $i+1$ bin and we sum the weight of the items collected in the i bin. If this sum is bigger than the capacity in at least one bin, we set $is\_feasible\_solution = 1$.

After we compute the total cost of the bins used by the new solution and we control if it is bigger than the max budget available and, if this is true, we set $is\_feasible\_solution = 1$.
The last check is done only when the size of the population is bigger than the parameter *perc_size_pop* defined in *main.py* to have more elasticity so, before this value, we accept also new solutions that don't improve the minimization of the problem to have a "*richer*" population.
In this part we control if the value of objective function of the new solution is bigger than:

a) the value of the objective function of the initial population if the dimension of population is $perc\_size\_pop + 1$;

b) the value of the objective function of the new solution computed in the previous step otherwise.

This division is done for a better improvement of the solution when the dimension size is bigger than $perc\_size\_pop + 2$.

4.3.3 change_random_gene

This function is created for the mutation operation. Its inputs are the population, the parameter *rand_index*, defined randomly in the function *mutate*, that is the solution of population selected for mutation, the vector *not_try* where we save the item tried, the parameter *it* used for appending element in *not_try* and *bin_el* that is the list of bin emptied in the *bin_mutation* function. The output is the population with the new solution in the last position.
In *change_random_gene* we select an item in the random solution found by *rand_index*, then we save the item in *not_try* and we create the new bin in a random way checking that the new bin isn't an element of the list *bin_el*. Until the old bin and the new bin for the item selected are the same we choose the new item always with a random function.
At the end there's a control for the item selected, in fact, the second time we use this function in *mutate*, we have to change the item until the new item selected isn't in *not_try* because otherwise we would have already used it before.

4.3.4 mutate

The aim of this function is to create a new feasible solution from a specific solution in the population changing randomly the bin that collects a par-

ticular item. The input is the population and the list of bin emptied in the *bin_mutation* function, and the output is still the population but in the last position there's the new solution with the mutated element.
First of all we choose randomly a solution in population with *rand_index* and we use the *change_random_gene* to mutate the bin that collects an item. This operation is also reused in a while cycle until:

- the function $is\_feasible$ returns 1, so the solution is unfeasible;
- the number of iterations is smaller than a fixed parameter *max_iter_while* defined in the *main.py*.

4.3.5 pick_two_parents

This function is used in the *crossover* function and its goal is to select randomly two solutions to which apply crossover and save their position of the population in a vector. The input is the population and the output is a vector of two elements respectively the first and the second position of the first and the second solution selected .
In particular, we randomly choose the positions, then we change their values with a while cycle until they are the same.

4.3.6 crossover

The aim of this function is to create a new solution applying crossover to two solutions of the population and append it to the population.
The input is the population and the output is the same as the input but with the result of the crossover in the last position.
First of all, we choose randomly a *point* between zero and the number of items where the crossover will happen. In a while cycle, with the same bounds of the function *mutate*, we create the new solution where the items from zero to *point* (excluded) are from the first solution selected and the other from the second one. Also we remember that the two solutions are choosen by the function *pick_two_parents*.

4.3.7 objective_function

The last function used by GA has the goal to compute the value of the objective function. The input are *solution*, n_b that is the number of bins, $cost\_b$ that is a vector of the cost of each bins, p_i that is a vector of the profit of each items, n_i that is the number of items and n_i_c that is the

number of compulsory items. The outputs are the value of the objective function and the vector of bins used, both related to the solution passed.
At the beginning we create the vectors *bins* of the bins used and the matrix *solution_matrix* whose dimension is (number of items, number of bins) and each position is 1 if a specific item is taken by that specific bin and 0 otherwise. Then we compute the value of the objective function in the same way of the $Next\_Fit$ and the $Tabu\_Search$ using the vector introduced here.

5 Results

In order to compare all the three methods described before we have created the script called *main.py*.
In here we have imported the instances and we have applied first the *Gurobi* method, then the $Next\_Fit$ algorithm, afterwords the $Tabu\_Search$ function and at the end the *Genetic_Algorithm* taking trace of the computational time of all the methods.
Before the *Genetic_Algorithm* we have defined three parameters:

- $size\_pop = 200$
- $perc\_size = 50$
- $max\_iter\_while = 100$

This parameters have been chosen because they are a trade off between accuracy and computational cost.

5.1 Comments

In the next pages we have reported five results for both the index's value.
As we can see Gurobi and Tabu Search have often similar final value of the objective function while the Genetic Algorithm always returns higher values and employs widely higher computational time.
Usually also the bins vectors are the same for Gurobi and the Tabu Search while the Genetic Algorithm are always different because are randomly generated.
About the computational time we can see that the Tabu Search always has a five or six times higher computational time than the Gurobi while the Genetic Algorithm has a computational time widely higher than the both of them.

Finally we have to recall that the computational time of Tabu Search and Genetic Algorithm in the next tables comprehends the Next Fit one.

Method	value of the objective function	bins	computational time
Gurobi	65,3904	[1,1,1,0,0]	0,073552s
Next Fit	232,9714	[1,1,1,1,1]	0,000021s
Tabu Search	100,7531	[1,1,1,0,0]	0,320499s
Genetic Algorithm	153,4909	[1,0,1,0,1]	185,511071s

Table 1: First iteration with index 0

Method	value of the objective function	bins	computational time
Gurobi	95,1495	[1,0,0,1,1]	0,146296s
Next Fit	215,0922	[1,1,1,1,1]	0,000882s
Tabu Search	33,0376	[1,0,0,1,1]	0,517553s
Genetic Algorithm	117,4362	[0,0,1,1,1]	209,081442s

Table 2: Second iteration with index 0

Method	value of the objective function	bins	computational time
Gurobi	25,1776	[0,1,0,1,1]	0,061711s
Next Fit	210,5172	[1,1,1,1,1]	0,000968s
Tabu Search	98,7824	[1,1,0,1,0]	0,351914s
Genetic Algorithm	112,8576	[0,1,0,1,1]	209,376728s

Table 3: Third iteration with index 0

Method	value of the objective function	bins	computational time
Gurobi	39,4075	[1,1,0,1,0]	0,054019s
Next Fit	158,1660	[1,1,1,1,1]	0,000998s
Tabu Search	-41,3415	[0,1,1,1,0]	0,398964s
Genetic Algorithm	80,2491	[1,0,1,1,0]	138,309013s

Table 4: Fourth iteration with index 0

Method	value of the objective function	bins	computational time
Gurobi	48,5445	[1,1,1,0,0]	0,057927s
Next Fit	205,3138	[1,1,1,1,1]	0,000997s
Tabu Search	127,2696	[1,1,1,0,0]	0,431690s
Genetic Algorithm	104,4298	[1,0,0,1,1]	119,765775s

Table 5: Fifth iteration with index 0

Method	value of the objective function	bins	computational time
Gurobi	106,8960	[1,0,1,1,0]	0,063908s
Next Fit	240,2135	[1,1,1,1,1]	0,000043s
Tabu Search	46,8121	[0,0,1,1,1]	0,364023s
Genetic Algorithm	164,2155	[1,1,0,0,1]	127,081202s

Table 6: First iteration with index 1

Method	value of the objective function	bins	computational time
Gurobi	-28,8521	[1,1,0,1,0]	0,062086s
Next Fit	166,3267	[1,1,1,1,1]	0,000009s
Tabu Search	8,6539	[1,1,1,0,0]	0,400618s
Genetic Algorithm	103,3267	[1,0,1,1,0]	118,915620s

Table 7: Second iteration with index 1

Method	value of the objective function	bins	computational time
Gurobi	-37,8565	[1,0,1,0,1]	0,064051s
Next Fit	90,8694	[1,1,1,1,1]	0,000015s
Tabu Search	-48,8565	[1,0,1,0,1]	0,413894s
Genetic Algorithm	26,8694	[0,0,1,1,1]	106,446434s

Table 8: Third iteration with index 1

Method	value of the objective function	bins	computational time
Gurobi	5,7116	[1,1,0,1,0]	0,172875s
Next Fit	202,8290	[1,1,1,1,1]	0,000012s
Tabu Search	14,8873	[1,0,1,0,1]	0,604413s
Genetic Algorithm	144,8290	[0,0,1,1,1]	110,995398s

Table 9: Fourth iteration with index 1

Method	value of the objective function	bins	computational time
Gurobi	0,1775	[1,1,0,0,1]	0,115021s
Next Fit	213,7876	[1,1,1,1,1]	0,001054s
Tabu Search	17,1775	[1,1,0,0,1]	0,609348s
Genetic Algorithm	152,7876	[1,0,1,0,1]	138,245393s

Table 10: Fifth iteration with index 1

5.2 Conclusion

For our problem the Gurobi methods and the Tabu Search Algorithm seem to perform widely better than the Genetic Algorithm, but it might be interesting to test these three methods with different instances, especially increasing the number of items and bins.
Another improvement for the Genetic Algorithm could occur changing the method used for the creation of the initial solution. In particular we have seen that if the bins weren't almost filled the Genetic Algorithm, due to the bigger room for manoeuvre, worked better.