An Heuristic approach to Generalized Bin Packing problem

Claudio Fantasia
Politecnico di Torino
Student id: s319911
s319911@studenti.polito.it

***Abstract*—The Generalized Bin Packing Problem is an extension of the classical Bin Packing Problem, in which the items are divided in compulsory to deliver and not strictly compulsory, while the bins have variable capacity and cost. The objective is to minimize the total net cost. This article presents two different heuristic approach to solve this optimization problem: an offline constructive heuristic and a Kernel Search Grouping Genetic Algorithm**

I. Problem overview

The Generalized Bin Packing Problem (GBPP) is an extension of the classical **Bin Packing Problem (BPP)** [1], where given two sets of compulsory and non compulsory items characterized by volume and profit, different type of bin's sets with given volume and cost, we want to select the subset of profitable non-compulsory items to be loaded together with the compulsory ones into the suitable bins in order to minimize the total net cost without exceeding the given budget. Throughout the years several formulations of the Bin Packing has been proposed. The most famous ones are the **Variable Size Bin Packing Problem (VSBPP)** and the **Variable Size and Cost Bin Packing Problem (VSCBPP)** [2].
These formulations are closer to the transportation field in these times. Furthermore the GBPP formulation is even closer to reality, because it takes into account the differences between items with more priority and the ones with less.
The mathematical model of this optimization problem is:

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

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

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

$$\sum_{j\in J} x_{ij} \le 1, \quad \forall i \in I^{NC} \tag{4}$$

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

$$\sum_{j\in J^t} y_j \le U_t \quad \forall t \in T \tag{6}$$

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

Where:

- J : set of bins
- I : set of items divided in compulsory item I^C and non compulsory item I^{NC}
- C_j : cost of bin j
- W_j : capacity of bin j
- p_i : profit of item i
- w_i : weight of item i
- $x_{ij} = 1$: if item i is delivered by bin j
- $y_j = 1$: if bin j is rented

The objective function (1) minimizes the total net cost of the package delivering, the first term represents the total cost of the used bins, while the second term stands for the total profit of the selected non_compulsory items. Constraint (2) force not to exceed the bins capacities. Constraint (3) force to select all the compulsory items and to allocate each item into one bin. This means that one item can't be delivered by more than one bin. While constraint (5) limits the budget that can be used. Instead (6) means that we can rent a limited amount of bin of a specific type.
It is important to notice that the profit of the compulsory items is not considered in the objective function because it is a constant.

II. Class Instances

The variables of our problem are instantiated in the following formulation:
n_items : $[50, 100, 150, 200]$
$w_i \sim Unif(1, 60)$, $w_i \sim Unif(1, 80)$, $w_i \sim Unif(1, 100)$
$compulsory\_percentage$: $[0.5, 1]$
$p_i \sim Unif(0.5 \cdot w_i, 3 \cdot w_i)$
$n\_type\_bin = 4$, $W_t \in [80, 120, 180, 250]$, $C_t \in [80, 120, 180, 250]$.
V_tot : sum of all the items
$U_t \in range(V\_tot/(W_t \cdot n\_type\_bin), V\_tot/W_t)$
$B = 0.7\cdot$ sum of all bin cost

These parameters lead to a few considerations: it does not exist a privileged bin that has good ratio C_t/W_t. There are not enough bins of one type to deliver all the items and the budget is limited meaning that it is not allowed to rent all the bins.

Furthermore it is important to notice that in the case where $compulsory\_percentage = 1$ the GBPP becomes a version of the classical VSCBPP.
Note that the vector containing our items is composed with compulsory items at top (i.e. [0, ... ,n_compulsory_items, ... ,n_items])

III. Offline Constructive Heuristic

In contrast to online algorithms, offline algorithms are designed to process an entire dataset at once [3]. These algorithms assume that all the dataset is available upfront and can be accessed in its entirety. It is assumed that this is the framework of our problem.
Given that the entire dataset has been sorted in the following way:
$Bins$: Non-decreasing ratio C_j/W_j. If two bins have equal ratio we sort by non-decreasing volumes W_j
$Compulsory\_items$: Non-increasing w_i
$Non\_compulsory_items$: Non-increasing ratio p_i/w_i. If two items have equal ratio we sort by non-increasing w_i

As it has been explained in the previous section, the compulsory items are on top and they are picked for first.

A. Main Procedure

The main variables of our heuristic are the sorted list of items (SIL), the sorted list of bins (SBL) and the list of bins rented (S) that will be updated during the algorithm.
We iterate over all the items in SIL and for each item we identify the bin from our list of bin rented (S) using a First-Fit(FF) or Best-Fit(BF) algorithm, and if we do not find any bin avaiable inside S, we search in SBL.
During our research in SBL, we have to take into account if we are treating compulsory item or a non compulsory one. If we are treating a compulsory item we are going to rent the first bin avaiable in SBL that respects the constraints (2). Instead if we are treating non compulsory item we apply the $PROFITABLE$ function to that item, to find a new bin to rent.
If we are not able to find the right bin using profitable function, we reject the item. At the end we perform a post-optimization phase

B. Profitable

The objective of this function is to understand if after locating the contested item and the consecutive items (without exceeding the bin capacity (2)) the profit of the items delivered will pay off the cost of the bin. If this is true, we can rent this bin and not reject the item.

C. Post-optimization

We iterate over the list of bin rented obtained by the main procedure. If we find a less expensive bin included in SBL that can deliver all the items located in one bin present in S, we choose to deliver these items using this another bin.

IV. Kernel Search Grouping Genetic Algorithm

The genetic algorithm often begins by creating a random initial population. In some scenarios it is preferable to start the initial population closer to the optimal solution.
For this reason it has been chosen to start the initial population using a Kernel Search [4]: the method is based on the identification of a restricted set of promising items and bins.

A. Initial Population

Solving the continuous relaxation of the problem using Gurobi, the solutions greater than 0.95 are inserted into the *Kernel Matrix*, while the solutions between 0.1 and 0.95 are considered as *Bucket solutions*. We create our initial population starting from the Kernel Matrix and adding one bucket solution at time, we obtain $num\_bucket\_solution$ elements in our population differing for only one element. If the population size is smaller than 200, we duplicate the population by an integer factor.

B. Group Encoding

Performing Genetic Algorithm(GA) on matrix is computational expensive and also inefficient. Given that it has been chosen a more compound form that takes into account the group structure oriented of GBPP [5]:
Given a vector $X \in R^{n\_items}$, we have $X(i) = j$ if item i is delivered by bin j, otherwise $X(i) = -1$ if item i is not delivered.
Using this encoding, one item can not be delivered by two bin at the same time breaking the constraints (3) and (4)

C. Fitness function

The fitness function has been set up to maximize the inverse of the objective function (1).
To prevent the algorithm from converging to an infeasible solution, a penalty factor has been incorporated into the solution value. Penalization factors are: 0.95 for each compulsory item not delivered, 0.85 for each bin capacity exceeded, 0.7 if the solution spend more than the budget.
Given the small weight that some items can assume, another constraint has been added: one bin is not able to deliver more than five items. This constraint should assist the *exploration phase* of the algorithm.

D. Crossover and Mutation

As a matter of fact the population is quite small (i.e ~ 200) and the solutions are all similar to each other. For this reason the $mutation\_rate$ has been set to 0.9. The highest mutation probability has been assigned to non compulsory items leading to a better exploration of the solution space.
Besides the mutation, for achieving more diverse generations we apply crossover rules to combine two parents and to form children for the next generation. It has been assigned probability 0.2 to *single-point crossover*, probability 0.1 to *two-point*

crossover [6], while probability 0.7 to not crossover at all. Notice that these values refer to a single child generation. The number of generation has been set to 150.

E. Post-Optimization

Despite the penalization in the fitness function, it can happen that the best solution found by the algorithm is infeasible. The solution is then fixed by the post-optimization function.
Iterating from the non compulsory items we search for the bin where the capacity has been exceeded, then we find the first item delivered by the aforementioned bin and we remove the item from the bin. We also try to replace the item with another item with less weight.This approach fix the problem of bin's capacity exceeded (2)
Instead for fixing the constraint (3), we first search for compulsory items not delivered, we try to accommodate the item in a bin that we have already rented, if the research is unsuccessful, we rent another random bin.

V. Results

For computing the exact solution for the problem instances Gurobi has been used. Given the fact that some instances are computational challenging the computation of the Gurobi's solution stops after finding a solution with a gap of 4%(with respect to Best Bound) or after 1000 seconds.
For instantiating the problem classes, numpy random seed has been set up to 42.
The CPU used was 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	0.57	> 1000	910	18.81
$w_i = (1, 80)$	1.97	> 1000	> 1000	> 1000
$w_i = (1, 100)$	34.24	> 1000	> 1000	> 1000

TABLE I: *Gurobi* results: $computational\, time(seconds)$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	180	112	195	495
$w_i = (1, 80)$	238	146	278	658
$w_i = (1, 100)$	295	192	331	820

TABLE II: *Gurobi* results: $objective\ function$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	0.028	0.078	0.126	0.32
$w_i = (1, 80)$	0.068	0.122	0.247	1.24
$w_i = (1, 100)$	0.023	0.315	0.77	2.82

TABLE III: *Gurobi* results: $computational\, time(seconds)$

(a) compulsory_percentage = 1

Considering the Gurobi's solutions as the Best Bound, the mean Gap along w_i has been computed.
For the accuracy of Offline Constructive Heuristic we have:
Gap for compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	1420	2990	4500	6120
$w_i = (1, 80)$	1890	3940	5940	8120
$w_i = (1, 100)$	2350	4900	7470	10130

TABLE IV: *Gurobi* results: $objectivefunction$

(a) compulsory_percentage = 1

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	0	0	0.0009	0.003
$w_i = (1, 80)$	0	0.0005	0.002	0.001
$w_i = (1, 100)$	0.001	0.0009	0.001	0.002

TABLE V: *Offline Constructive Heuristic (Best-Fit)* results: $computational\, time(seconds)$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	227	248	287	586
$w_i = (1, 80)$	367	334	463	860
$w_i = (1, 100)$	464	525	713	1212

TABLE VI: *Offline Constructive Heuristic (Best-Fit)* results: $objectivefunction$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	0	0	0.001	0.0009
$w_i = (1, 80)$	0	0	0.001	0.0009
$w_i = (1, 100)$	0.001	0.001	0.0008	0.001

TABLE VII: *Offline Constructive Heuristic (Best-Fit)* results: $computational\, time(seconds)$

(a) compulsory_percentage = 1

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	1420	2920	4440	6040
$w_i = (1, 80)$	1960	4000	5920	8160
$w_i = (1, 100)$	2440	5040	7720	10500

TABLE VIII: *Offline Constructive Heuristic (Best-Fit)* results: $objectivefunction$

(a) compulsory_percentage = 1

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	2.15	4.17	6.54	10.96
$w_i = (1, 80)$	2.42	4.38	8.12	9.55
$w_i = (1, 100)$	2.52	4.72	8.06	8.90

TABLE IX: *Kernel Search Grouping GA* results: $computational\, time(seconds)$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	535	1093	1255	2767
$w_i = (1, 80)$	742	1219	1687	5290
$w_i = (1, 100)$	1042	1666	3249	4210

TABLE X: *Kernel Search Grouping GA* results: $objective\ function$

(a) compulsory_percentage = 0.5

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	2.26	4.38	6.26	10.34
$w_i = (1, 80)$	2.35	4.56	7.31	8.65
$w_i = (1, 100)$	2.38	5.28	8.30	8.93

TABLE XI: *Kernel Search Grouping GA* results: $computational\ time(seconds)$

(a) compulsory_percentage = 1

	$n = 50$	$n = 100$	$n = 150$	$n = 200$
$w_i = (1, 60)$	1860	3330	4830	6530
$w_i = (1, 80)$	2000	4520	6280	9360
$w_i = (1, 100)$	2580	5490	8860	11120

TABLE XII: *Kernel Search Grouping GA* results: $objective\ function$

(a) compulsory_percentage = 1

$n = 50 : 45\%$,$n = 100 : 141\%$,$n = 150 : 76\%$,$n = 200 : 32\%$
Gap for compulsory_percentage = 1
$n = 50 : 2.5\%$,$n = 100 : 0.67\%$,$n = 150 : 0.55\%$,$n = 200 : 0.94\%$

For the accuracy of Kernel Search Grouping GA we have:
Gap for compulsory_percentage = 0.5
$n = 50 : 220\%$,$n = 100 : 792\%$,$n = 150 : 643\%$,$n = 200 : 525\%$
Gap for compulsory_percentage = 1
$n = 50 : 15\%$,$n = 100 : 12\%$,$n = 150 : 10\%$,$n = 200 : 10\%$

VI. Discussion

We can see that the easier class instantiation is the one with $compulsory\_percentage = 1$, because the computational cost seems to be low for all the algorithms and the accuracy seems to be better for both the Heuristic Algorithms.
The offline Constructive Heuristic **(OFH)** has way better results than the Kernel Search Grouping Genetic Algorithm **(KSGGA)**, undoubtedly the (OFH) offers good solutions with small Gap with respect to the Gurobi's solution in a small amount of time. The best solutions has been obtained with compulsory_percentage = 1, but we have to notice from table IIIa that the Gurobi's solutions have fast convergence in this case and it is quite pointless to use inexact method.
It's undeniable that (KSGGA) has poor performance. But it's worth notice that the (KSGGA) solution is often similiar to the first heuristic solution found by Gurobi(using less computational time).
For boosting the performance of (KSGGA), it was tried to adjust the solution during fitness evaluation when it broke the constraints, but this led to poor convergence.
It was also tried to increase the population size to 8000, increasing the duplicating factor of bucket solutions and injecting some random value between (0,n_bins) in order to help the exploration of the algorithm. But this led to higher computational cost and lower accuracy.
One of the main problems of (KSGGA) is that most of the results before post-optimization function are infeasible solutions and after post-optimization the objective function has way less performance than the previous one.

In Conclusion, Kernel Search mixed with Genetic Algorithm is probably not a good choice. Perhaps for exploiting better the Kernel solution it is better using Tabu search algorithm that may have better exploitation of the solution.

References

[1] M. M. Baldi, "The generalized bin packing problem," 2012.
[2] M. M. Baldi, *Generalized Bin Packing Problems*. PhD thesis.
[3] E. G. Coffman, "Approximate solutions to bin packing problems," 1999.
[4] E. Angelelli, "Kernel search: A new heuristic framework for portfolio selection," 2012.
[5] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," 1996.
[6] M. H. a, "Heuristics for the variable sized bin-packing problem," 2009.