A hyper-heuristic for solving variants of the 2D bin packing problem

Alkiviadis Aleiferis

alkiviadis.aliferis@gmail.com
19 September 2023, Athens, Greece

1. Introduction

The current work aims at solving a set of variants of the two-dimensional bin packing problem (2DBPP). The plethora of variants arises due to the dynamic nature of the algorithm to abstract the solution flow towards different variants, without disrupting the process. The type of problems I'll be addressing have all the below characteristics:

A. Are two-dimensional (geometrically).
B. Have any variation of given rectangular small items.
C. Any number and size of rectangular large objects (bins), although predetermined.
D. Small items can be rotated or not.

Firstly, a constructive heuristic will be described for creating a feasible solution through a deterministic algorithm, provided a given sequence of items. The appointed name of the method is "Point Generation", since it relies on the generated points of possible item placement to move towards the solution.

Secondly, a non-learning hyper-heuristic method will be presented, utilizing a hill climbing local search metaheuristic for traversing neighborhoods of solutions towards local optima, and a generative alteration of the heuristic to push the search towards other regions of the solutions space.

The above methods will be implemented in Python, and will become an openly distributed python package through the official PyPi repository, named "hyperpack".

2. Point Generation constructive method

2.1 Overall description

The overall method can be summarized into an algorithm, that takes as **input** a sequence of items **S** (each item has it's own id) whose length equals the cardinality of the total items set **I** to be stored, and using a deterministic method of placing them sequentially on **potential points P** inside the **bin B**, and generating those said points in the process, produces an **output** of the packed bin, with the placing coordinates (X_o, Y_o) of every item, and the info about their rotation state (if rotation is enabled) inside the solution. In figure 1 we can see the bin B and its coordinate system and dimensions W (width) and L (length).

The **key concepts** in this process are the generated **potential insertion points,** denoted by **P**, of item placement (points where an item can be inserted in the bin, with its origin point), and the *horizontal* and *vertical* **line segments** produced while constructing the solution, denoted by **H**, **V,** after each placement of an item. Each line segment is depicted as **a set of points**, mathematically speaking the two edges of the segment in Cartesian coordinates $\mathbf{((x_1, y_1), (x_2, y_2))}$.

The item's **origin point** is it's bottom left corner, with coordinates (X_o, Y_o), which is also the **insertion point**. In case rotation is enabled, the insertion point remains the same, but the item's width and length dimensions are interchanged in the item set. This can be seen in figure 2. Points **O**, **A** and **B** are used for quick reference in the sections below, as the item's upper left and bottom right corners.

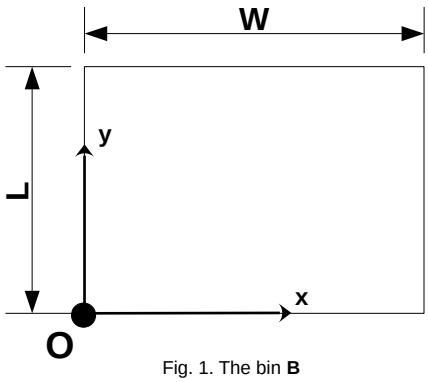

Fig. 1. The bin **B**

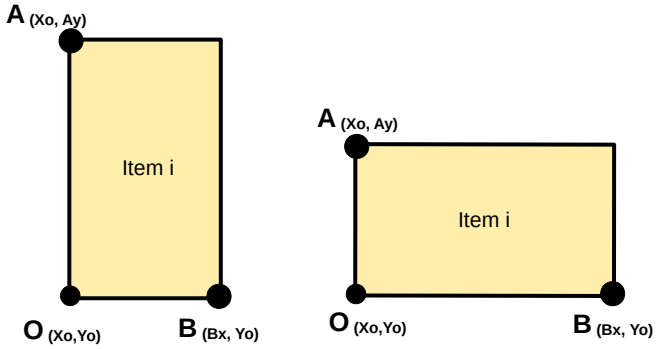

Fig. 2. Item's rotation and insertion point

The process starts from an initial potential point (origin O(0, 0) of the bin), so P_0 = {(0, 0)}, and places item after item (from the items sequence pool) in the bin on sequentially generated potential points (avoiding overlapping through proper checks), and after each placement the new item generates new potential points for other items to be inserted, according to certain rules, and appends it's line segments (four of them due to rectangular shape) on the list of vertical and horizontal lines. Before the insertion the **overlapping constraints** are deployed against every item already in the bin to ensure non-overlapping between items, thus insertion feasibility. The line segments are used by the potential point generation technique, and the potential points for the item placement. After each insertion at a potential point, the point used is removed from the pool of available points. The **termination state** is either the empty pool of potential insertion points, or the empty pool of items to insert.

At the start, the algorithm possesses the four segments of the bin: **V** = {((0, 0), (0, L)), ((W, 0), (W, L))} are the initial vertical segments of the bin and **H** = {((0, 0), (W, 0)), ((0, L), (W, L))} are the corresponding horizontals, where **W, L** are the width (X coordinate) and length (Y coordinate) of the bin. Each inserted item appends its sides as segments with the corresponding starting and finishing points. Due to the rectangular shape, it's obvious that the number of segments is four.

The potential points are classified in certain categories. In general we have the classes A, A', A", B, B', B", C, D, E, F. The classification serves as we 'll see later, the direction of general insertion. Explanatory figures are given below for each potential point class.

Notation

- S, the sequence of items,
- P, the potential points set,
- I, the items set,
- H, V, the horizontal and vertical segments,
- W, L, the width and length of the bin,
- w, l, the width and length of the item,
- A, B, the upper left and the bottom right corner of the item,
- A_Y, B_X, the y coordinate of the item's A point and the x coordinate of the item's B point.

2.2 Overall solution Algorithm

Input:
  items ← {item_id: {w: int, l: int},...}
  containers ← {container_id: {W: int, L: int},...}
START
objective_value_per_container = {}
solution ← {}
for container_id, container in containers:
  solution[container_id] ← {}
  objective_value_per_container[container_id] ← 0
  if items is empty:
    continue
  end if
  new items, objective_value, solution[container_id] ← **Point generation heuristic**(container, items)
  objective_value_per_container[container_id] ← objective_value
end for
return objective_value_per_container, solution
END

2.3 Point generation heuristic Algorithm

Input:
  items ← {item_id: {w: int, l: int},...}
  container ← {W: int, L: int}
  potential_points_strategy (sequence of preferred points classes)
  rotation ← True or False
START
items_sequence ← items ids in sequential form
objective_value ← 0
potential_points ← initial potential points (Only bin's Origin O (0, 0))
current_solution ← {}
H ← the bottom and top side of the bin
V ← the left and right side of the bin
current_point, point_class ← O (0,0), 'O'
while (items not empty) **or** (objective_value < 1) **or** (potential_points not empty)
  X_0, Y_0 ← current_point's coordinates (insertion point)
  # first fitting item in the items sequence gets in the solution
  for item in items
    check ← check if item fits
    if not check
      if rotation enabled:
        check ← check if item fits
      if not check:

```
            continue to next item
          end if
        else
          continue
        end if
      current_solution[item_id] ← X0, Y0, w, l, rotation
      remove item from items
      objective_value ← objective_value + (w*l)/(W*L)
      A, B ← (X0, Y0 + l), (X0 + w, Y0)
      potential_points ←append generated points from new item
      H, V ← append segments generated from new item
  end for
  # get next insertion point from the pool, if pool not empty
  # use the potential_points_strategy to pick the first point, from the
  # first non empty preferred points class
  current_point, point_class ← get current point using potential_points_strategy
end while
return items, objective_value, current_solution
```

2.4 A, B potential points classes

In figure 3.1 one can see and intuitively understand points A and B. First item ("item 0") of the items sequence is always placed at the bottom left corner of the bin (Origin O(0, 0)). This placement generates points points A_0 and B_0 of the first item and appends them on the list of A and B points correspondingly. These points are characterized as first class, due to the fact that if a solution is able to insert all the items to potential points of those classes, a global optimum is most probable to be achieved due to the fact that insertion on those points leave no space between tangential items.

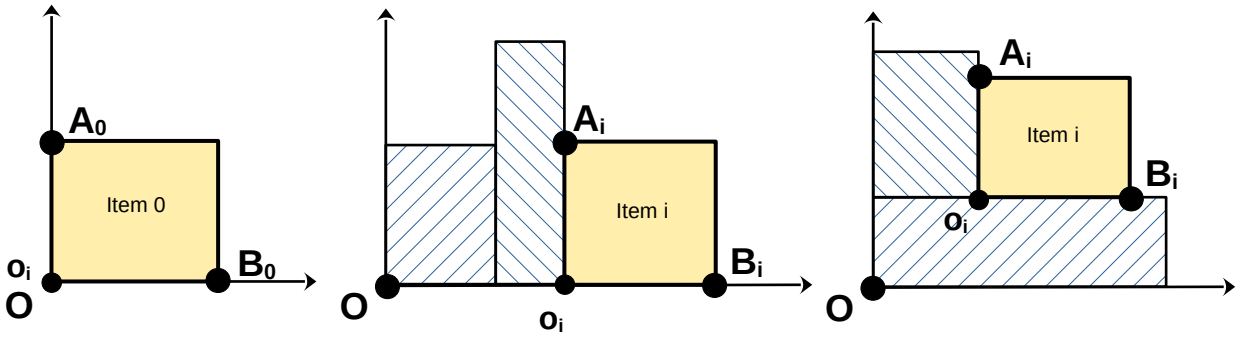

Fig. 3.1. A,B potential points locations

The **rule of generating points A_i** is the existence of a vertical segment at placement, such that:

- The X coordinates of the segment (they are both equal due to verticality) is equal to the X coordinate of the A_i point (X_0).
- The Y coordinates of the said segment completely engulf the A_Y coordinate.
- If X_i is 0 and $A_Y < L$, then A_i is automatically appended to the list of A points.

The vertical segment can be any of the already existing up to the moment of placement, either the bin's initial ones, or the ones appended by the until then inserted items.

The **rule of generating points B_i** is the existence of a horizontal segment at placement, such that:
- The X coordinates of the horizontal segment completely engulf the B_X coordinate.
- The Y coordinates of said segment (they are both equal due to horizontality) is equal to the Y coordinate of the B_i point (Y_0).
- If Y_0 is 0 and $B_X < W$, then B_i is automatically appended to the list of B points.

The horizontal segment can be any of the already existing up to the moment of placement, either the bin's initial ones, or the ones appended by the until then inserted items.

The above rules have **edge cases** that need certain conditions to be checked. In figure 3.2 we can visualize these predicaments.

In figure 3.2 one can see the restrictions of generation in those cases. In edge of the bin placements where A touches the far edge horizontal segment of the bin, or B touches the far edge vertical one, these points will not be appended in the potential points pool.

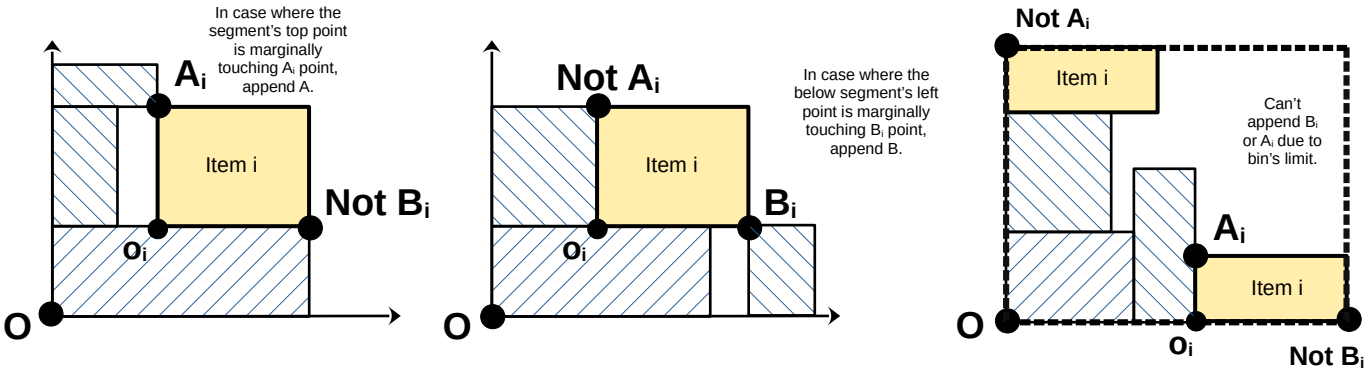

Fig. 3.2. A,B potential points marginal cases

2.5 A', B' potential points classes

In this case, the points to be generated arise from the situation, where either the *corners A_i or B_i do not come in contact with either the bin's or another items vertical or horizontal segment*, thus unable to satisfy the A and B points' rules of generation. Basic prerequisite for these points is that no A, B points where generated.

In figures 4.1 and 4.2 it can be seen clearly that in these cases we try projecting the A_i and B_i point horizontally or correspondingly vertically towards the first line segment, that the "projection" can "land". Mathematically, for A' points, this means the first vertical segment with top point's Y coordinate greater than A_Y and bottom point's Y coordinate lesser than A_Y. We follow the same logic for B' points for "landing" on horizontal segments.

A certain rule for this point is the number of vertical segments with top Y coordinate in between the item's left side's Y coordinates, that is the segments that exists between the item's left side and the landing segment, which for reference we 'll call **intersegments** *(can be seen with green color on the figures)*. When this number increases, so does the number of A and A' points that could have been otherwise utilized by bigger items, now are forced

towards tighter insertion constraints. This creates a differentiation that has to be interpreted in terms of potential points classes due to optimization reasons.

Following this logic we appoint the A' point if the number of traversed in between (including the landing) segments are less or equal to two. Otherwise we append an E class point. We have exactly the same logic in the case of B' points, but the projection follows a vertical path towards a horizontal line segment, and the alternative point for greater than two intersegments is the F class point.

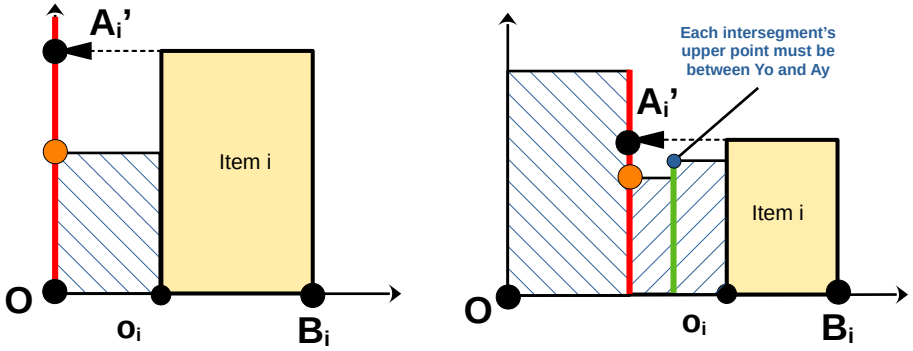

Fig. 4.1. A' potential insertion points

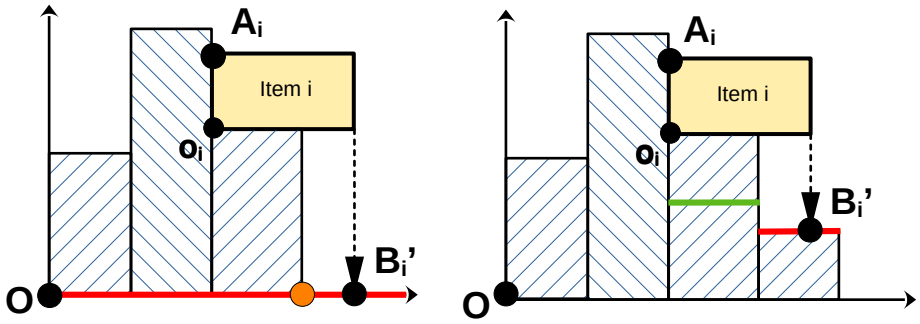

Fig. 4.2. B' potential insertion points

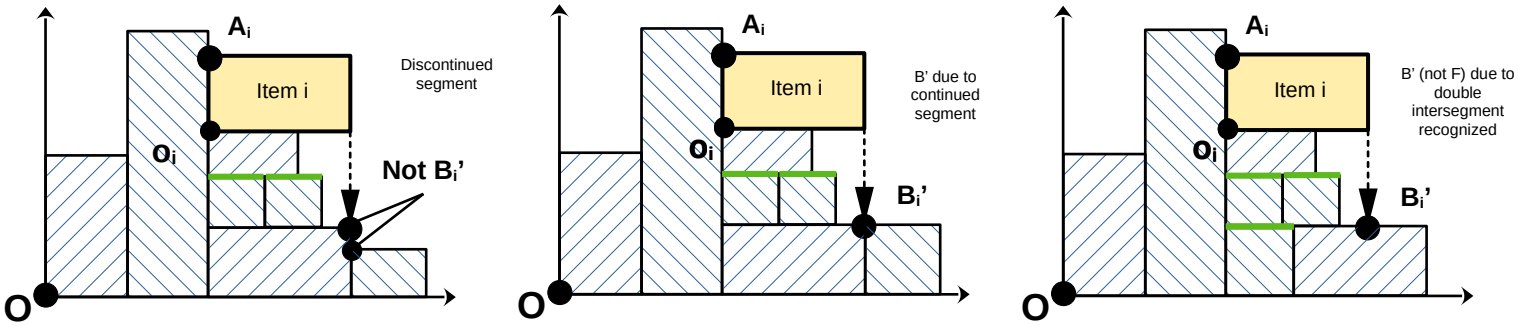

Fig. 5. A' and B' potential insertion points marginal cases

As with the first classes, we have some edge cases to consider. These can be seen in figure 5. Starting with the A' class, we have the following marginal cases:

- More than one intersegments exist at one X coordinate. In this case the algorithm must avoid counting both those segments as intersegments, thus only one intersegment can exist per X coordinate.
- when A' projection lands on another items bottom right corner (segment's starting point). In this case we add the A' point to the pool of A' points.

- when A’ projection lands on another item’s top right corner (segment’s ending point). In this case we avoid adding the A point to the pool of A’ points, if the line isn’t continued by another segment.

For B’ class edge cases we follow the same pattern, prohibiting creation of B’ points on corners of segments’ ending points that aren’t continued and avoiding to enumerate more than one intersegments at the same Y coordinate.

At both point generation situations when we have an exact obstruction of the items A or B point, the creation of the A’ and B’ (also E and F) points is not allowed.

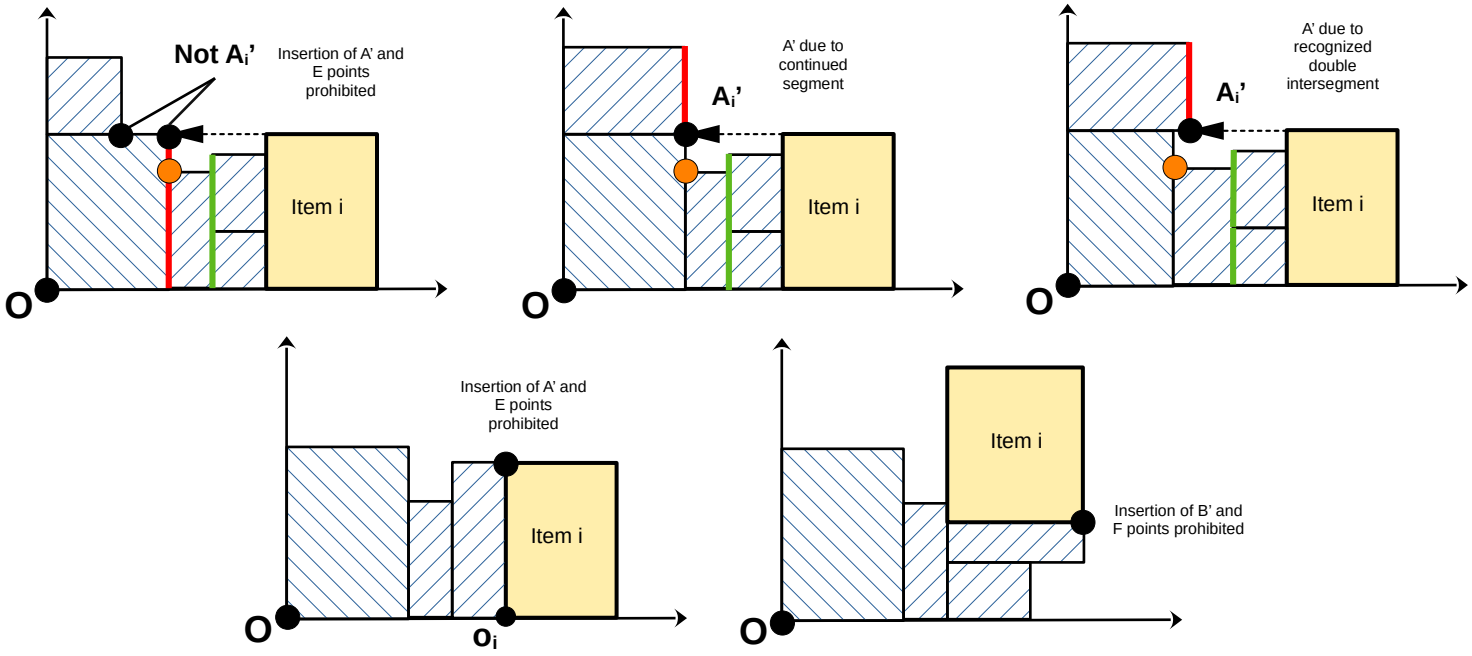

Fig. 5 (continued). A’ and B’ potential insertion points marginal cases

2.6 E, F potential points classes

The E and F classes are an extension of the A’ and B’, with the only difference that if more than one intersegments exist between A or B point and the landing segment, then the A or B point is appended to the E and F class insertion points pool correspondingly. Figure 6 depicts such cases. Marginal cases between A’ and E, and B’ and F, are common.

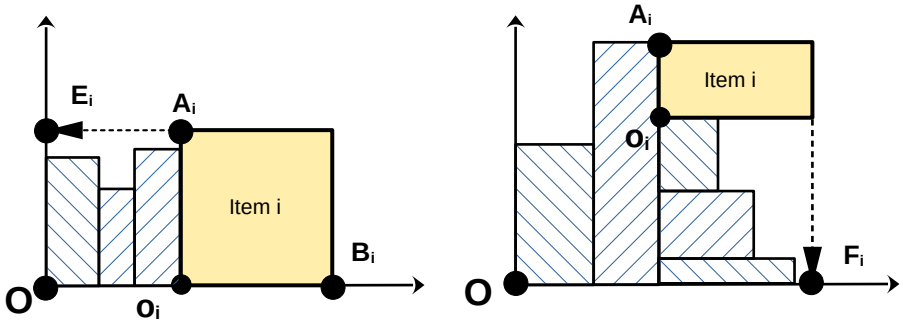

Fig. 6. E and F potential insertion points

2.7 C, D potential points classes

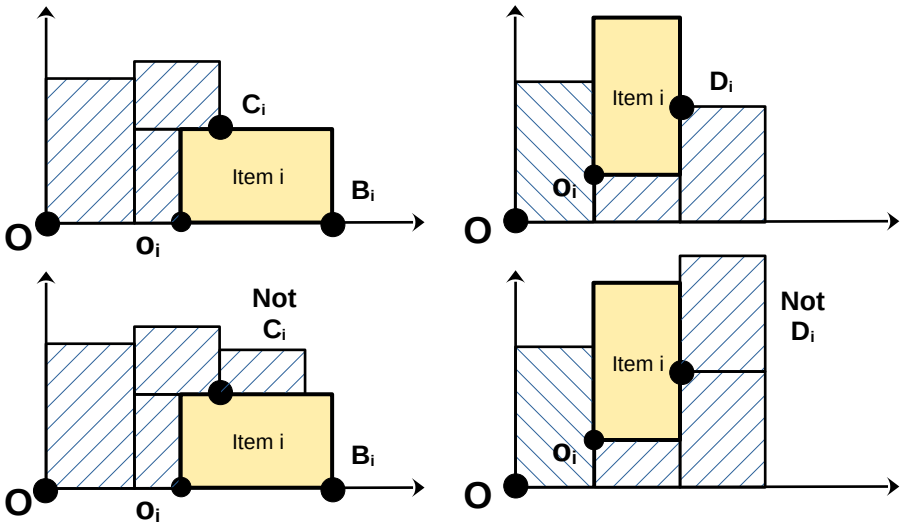

Fig. 7. C and D potential insertion points

In this case another item is found already “touching” on one of the upper or right item’s segment (not edge), thus leaving a potential point exposed. We denote those points with C and D as one can see in figure 7.

The rules for C points are:

- The on top item’s bottom segment ending X coordinate, must be between X_0 and B_X.
- No other horizontal segment on the A_Y level must exist with starting X coordinate equal to the found C points X coordinate.

D points follow the exact same logic, but on the right side of the item, involving vertical segments.

2.8 A”, B” potential points classes

These classes are final standings for insertions that otherwise failed in any way, but an insertion could still benefit the solution even in these circumstances. They are A and B points, but without meeting the generation rules of A and B. These points are appointed under the below circumstances:

- A or corresponding B points haven’t been generated.
- $A_Y < L$ ($B_X < W$).
- No horizontal or vertical segment passes through A_i or B_i correspondingly, thus making the insertion infeasible on these points.

2.9 Horizontal and Vertical segments

The initial segments as mentioned before, are the bin’s sides. That is the segments with (starting point, ending point):

Horizontals → ((0, 0), (W, 0)), ((0, L), (W, L))

Verticals → ((0, 0), (0, L)), ((W, 0), (W, L))

where W, L are the width and length of the bin.

After each item placement, it's sides are appointed in those segments, where (X_0, Y_0) is the insertion point, $B_X = X_0 + w$ (=item's width), $A_Y = Y_0 + y$ (=item's length).

Horizontals → $((X_0, Y_0), (B_X, Y_0))$, $((X_0, A_Y), (B_X, A_Y))$
Verticals → $((X_0, Y_0), (X_0, A_Y))$, $((B_X, Y_0), (B_X, A_Y))$

2.10 Turning Potential points to insertion points (potential points strategy)

At the beginning of each insertion, we need to specify the insertion point. As described above, the pool of potential points is given by the classes described above. But the question arrives: Which points class do we choose each time? Thus we introduce the second variable component of the heuristic (first was the items' sequence), the potential points strategy, that is the sequence of those points classes pools that we utilize for insertion.

By example, the default strategy for the heuristic is → (A, B, C, D, A', B', B", A", E, F). That means if the A class points is empty, use the available from B class for insertion. If those are empty too, go to C class etc. If all points' classes pools are empty, no more insertion points are available, thus the algorithm concludes it's operation and returns the solution.

3. Hyper Search hyper-heuristic

3.1 Introduction

The constructive heuristic depicted on section 2, can create deterministic solutions for the same input, but introduces variability on two dimensions:

A. The item's sequence.
B. The potential points strategy.

Thus a search operation can be deployed for both those dimensions, although the first is **input** variability, while the second introduces **variability on the heuristic** itself.

The items' sequence input variability will be explored using a 2-opt hill-climbing local search for exploring all the neighbors of the initial sequence, and using the construction heuristic we find the solution's objective value for every sequence. The one that results to better objective value will be the next node for searching, until a node with no better neighbor is found. This process is mandatory due to the fact that parts of the solution (item placements) cannot be interchanged like the traveling salesman's problem solution components. Thus a construction heuristic should be deployed for every exchange in the input sequence of items.

Now the hyper heuristic is introduced through the alteration of the constructive heuristic's potential points strategy, ergo altering the heuristic itself used in local search. That way the search is raised a level higher to the heuristic space, above the solutions one.

The hyper heuristic doesn't utilize performance or data of previous potential points strategies searches to guide the search, thus it's a **non-learning** hyper heuristic.

Each iteration is deploying the local search from the same initial items' sequence, pushing the search to other solutions, through the alteration of the construction heuristic. The construction heuristic's strategies (component of the heuristic) are a predetermined set of choices to be traversed.

An exhaustive analysis and relation between items/bins populations characteristics and potential points strategies won't be explored.

The predetermined set of strategies to be searched are the total permutations of the potential points classes. For the A, A', A", B, B', B", C, D, E and F classes a total of 3,628,800 exist, but a subset of points permutations will be chosen due to probable lower impact on solution quality of the A", B", E, F points. That leaves the permutations of A, A', B, B', C and D points and a suffix of each strategy of A", B", E, F potential points sequence. That is a total of 720 strategies to be searched. *In a multiprocessing division of calculation labor the number of strategies is divided by the available processor threads*.

3.2 Local search algorithm

Input:
initial node (items sequence)
containers
potential points strategy
START
initial solution
best_solution ← Constructive Heuristic(potential_points_strategy, initial_items_sequence)
best_objective_value ← initial solution's objective value
current_node ← initial_items_sequence

if containers number > 1
 # that way we can make sure the first containers
 # are best utilized
max_objective_value ← containers number – 0.3 (symbolic <1 number)
else
max_objective_value ← 1
end if

 # while the best possible value hasn't been found
while (best_objective_value < max_objective_value):
neighbor_found ← False
for every neighbor of the current_node (2-opt exchange):
current_items_sequence ← neighbor
new_solution ← Constructive Heuristic(potential_points_strategy, current_items_sequence)
new_objective_value ← new solution's objective value
if new_objective_value > best_objective_value:
best_solution ← new_solution
best_objective_value ← new_objective_value
neighbor_found ← True
current_node ← current_items_sequence
break
end if

```
  end for

  # if a neighbor wasn't found, we have local optimum
  # local search exits
  # if found, start local search from new node
  if not neighbor_found:
    break
  end if
end while

return best_solution,  best_objective_value
END
```

3.3 Hyper heuristic algorithm

```
Input:
  initial items sequence
  containers
  potential points strategy
START
sorted_items_sequence ← Make any initial sequence sorting (e.g. increasing volume etc.)
best_solution ← Constructive Heuristic(default potential points strategy, sorted_items_sequence)
best_objective_value ← initial solution's objective value
best_strategy ← default potential points strategy
potential_points_strategies ← determine all the strategies to test for
max_objective_value ← same as local search

for strategy in potential_points_strategies:
  new_solution ← local search(strategy, sorted_items_sequence)
  new_objective_value ← new solution's objective value
  if (new_objective_value > best_objective_value):
    best_solution ←  new_solution
    best_objective_val ← new_obj_val
    best_strategy ←  strategy
  end if
  if  (best_objective_value >= max_objective_value):
    break
  end if
end for

return best_solution,  best_strategy
END
```

3.4 Initial items sequence sorting and orientation

The initial sorting of the items can exhibit a big influence on the best objective value. Throughout research an initial sorting has been proposed on the items sequence of placement, such as sorting according to perimeter, width to length ratio, surface etc. There won't be any implementation of mining of the relation between items sorting and solution speed or quality in this document.

4. Benchmarks

Since the theory provided here is implemented in a python library named 'hyperpack' accessible through the official python repository (PyPi), a benchmark is available to everyone with basic knowledge of Python. A good reference point for performance is the "An Empirical Investigation of Meta-heuristic and Heuristic Algorithms for a 2D Packing Problem" [1] article of E. Hopper and B. C. H. Turton. Inside the package benchmark data of this article are included [2]. Below are the results for the five first categories. The hardware used is i7 – 7700 CPU (8 threads) and 16GB RAM 3200 MHz. It would take approximately half time for a 16 threaded CPU and similar processing speed per thread.

Category 1 (C1) Container 20×20

Problem	**1**	**2**	**3**
Items number	16	17	16
Total execution time [sec]	0.9161	11.0126	0.7878
Bin utilization	100.0000%	99.0000%	100.0000%
Remaining items	0	1	0

Category 2 (C2) Container 40×15

Problem	**1**	**2**	**3**
Items number	25	25	25
Total execution time [sec]	1.0399	1.018	0.8794
Bin utilization	100.0000%	100.0000%	100.0000%
Remaining items	0	0	0

Category 3 (C3) Container 60×30

Problem	**1**	**2**	**3**
Items number	28	29	28
Total execution time [sec]	25.3497	129.902	29.9735
Bin utilization	100.0000%	99.7778%	100.0000%
Remaining items	0	1	0

Problem 2:
***Note**: the algorithm searched all the potential points strategies (720). Reached 99.1677% in first 4 seconds.*

Category 4 (C4) Container 60×60

Problem	1	2	3
Items number	49	49	49
Total execution time [min]	18.8	2.83	19.2
Bin utilization	99.7778%	100.0000%	99.1367%
Remaining items	1	0	1

Problem 1 note: the algorithm searched all the potential points strategies (720). Reached 99.1389% in first 8 seconds.

Problem 2 note: the algorithm searched all the potential points strategies (720). Reached 99.7778% in first 8 seconds.

Problem 3 note: the algorithm searched all the potential points strategies (720). Reached 99.5833% in first 8 seconds.

Category 5 (C5) Container 60×90
These problems will be subjected to one minute solving time.

Problem	1	2	3
Items number	73	73	73
Total execution time [min]	1	1	1
Bin utilization	99.5926%	100.0000%	99.6667%
Remaining items	1	0	1

Category 6 (C6) Container 80×120
These problems will be subjected to one minute solving time.

Problem	1	2	3
Items number	97	97	97
Total execution time [min]	1	1	1
Bin utilization	99.2917%	99.5833%	99.3854%
Remaining items	6	1	5

4. Conclusions

In the current document an approach for solving the 2D binpack problem has been made. Future research can focus on aspects of mining between the potential points strategies and items/bins population characteristics, as well as more innovative ways of utilizing the Point Generation heuristic in hyper-heuristic methods.

References

1. E. Hopper and B. C. H. Turton, 2000. An Empirical Investigation of Meta-heuristic and Heuristic Algorithms for a 2D Packing Problem. European Journal of Operational Research 128/1, 34-57.
2. Github hyperpack's repository: https://github.com/AlkiviadisAleiferis/hyperpack