Local Search and Meta-Heuristics Lab Problem

COMP4691-8691

The problem considers a set of n rectangular boxes that are to be packed into a set of m containers. Each box and each container has an x-length and a y-length. Furthermore each box as an associated weight w that represents how important it is that the box is packed into a container. A box can either be packed vertically or horizontally into the container. The aim of the problem is to pack as many boxes into containers as possible such that the sum of the weights of packed boxes is maximised. Note that boxes cannot overlap and must stay within the boundaries of the containers. A solution to a problem with four containers and 100 boxes is visualised in figure 1.

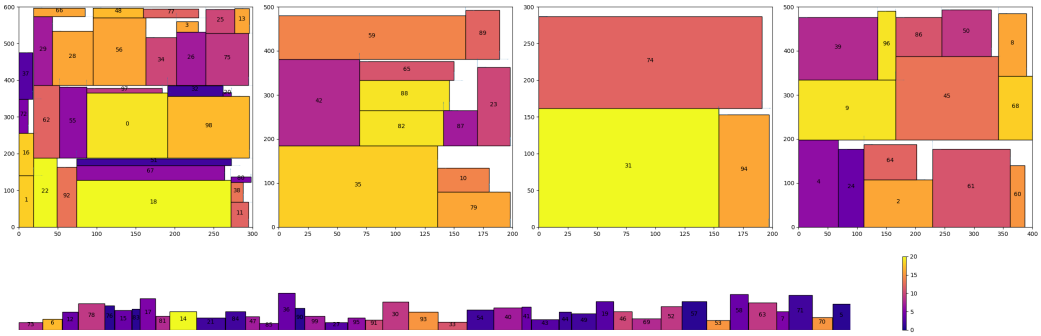

Figure 1: A solution for an instance with $n = 100$, $m = 4$.

1 Corner Heuristic

An efficient heuristic for the 2D packing problem can be developed that is based on a couple of insights: we may as well pack in corners, and corners are easy to track.

To do so, for all empty containers we only consider the bottom-left corner $(x, y) = (0, 0)$ as the initial corner. When we insert a box into that container we check whether the box can be placed (both horizontally and vertically) into that corner. A box can fit into a specific corner in a container in a specific direction if firstly the box fits inside the container, and secondly it does not overlap any other boxes that are already placed in the container. Once a box is inserted into a corner we can create two new corners - one at the top-left corner of the box, and one at the bottom-right corner of the box. In figure 2, we see how the corners progress as boxes are inserted into a container.

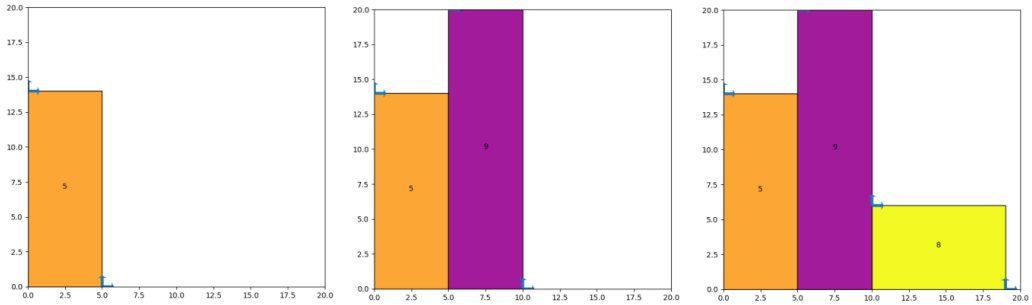

Figure 2: Inserting 3 boxes into a container with corner heuristic.

If a box cannot fit into any of the corners from any of the containers then in the final solution it remains unpacked. Note that the order that both the boxes and containers are considered by the heuristic will most likely produce different solutions, as well as the order corners are considered by a box and whether the box first attempts to be inserted horizontally or vertically.

1.1 Check Fit

Complete the `check_fit` function in the `corner_heuristic.py` file. This funciton should check whether a box with a particular orientation (i.e. either horizontal or vertical), can be inserted into a specific corner of a container.

Note: a position is infeasible if any part of the box is outside the container or if the box overlaps with any other box that is already positioned in the container.

1.2 Heuristic Algorithm

Implement the `corner_heuristic` function in the `corner_heristic.py` file. You should parameterise this function, so that the heuristic can utilise different orders for consideration of the boxes, containers, container corners and box orientations. The provided code sets you up to do this by passing lambda functions that take a list of the appropriate type, and return a new list in a particular order. Their default value is to just return the original ordering, but in the next question you will experiment with different ordering functions.

1.3 Experiments

Play around with different order functions for the corner heuristic on instance `case-1-20.json`. For the best combination you come up with, plot the solution you achieve, and state the objective it achieves. Discuss the ordering combinations you tried, and what the best one was you found.

Note: different orderings you could consider passing to this heuristic include: reversing the order, randomised order, sorted based on identifier, sorted based on weight, sorted based on size (container or box), sorted based on the remaining space in container. The `random.sample` and `sorted` python functions might come in useful for this.

For this instance, I can achieve an objective of 189 by ordering the boxes by largest to smallest weight. Figure 3 shows the result.

1.4 Larger Instances

For the same parameter configuration (no need to retune for each instace), use the corner heuristic to solve instances `case-4-100.json`, `case-5-200.json`, and `case-5-300.json`, and present the objectives achieved in a table alongside the `case-1-20.json` result.

What is one way you could know if you have achieved an optimal solution for any of these instances? Have you achieved an optimal solution for any of these instances?

The following table shows the results for the 4 instances:

Case	1-20	4-100	5-200	5-300
Objective	189	785	1664	3137

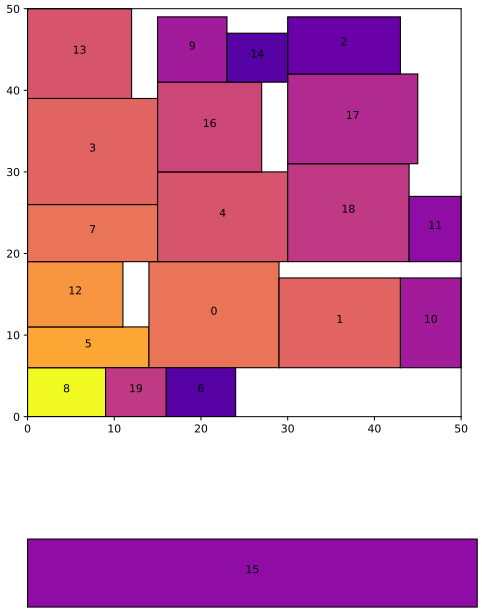

Figure 3: Solution for `case-1-20.json`.

If all boxes can be packed, then we know we have an optimal solution. Yes we achieved an optimal solution of 3137 for the `case-5-300.json` instance because all boxes were packed.

1.5 Incompleteness

In general when building a meta-heuristic it is desirable to ensure that the heuristic being used to generate solutions is complete, i.e. for a given parameter setting the heuristic will generate the optimal solution. The corner-heuristic is known to be incomplete. This means that for some problems, there does not exist a parameter setting (in terms of orderings) to the heuristic that will generate an optimal solution.

Provide a minimal counter example that demonstrates that the corner heuristic is incomplete, i.e. an instance of the 2D packing problem described

in this problem set, where no matter what parameters the heuristic is given it will never find the optimal solution. Briefly explain your answer.

A minimal counter example is based on a container with x- and y-lengths of 10, and four identical boxes with x-length 4, y-length 6, and equal weight. The reason why the original corner heuristic is not able to generate this solution is as follows. Due to the fact that all boxes are identical, the dimensions of the container are the same length, and taking into account symmetries, without loss of generality we can insert an box into the bottom-left corner of the container horizontally. It is possible to insert a box vertically at both the corners created by the first box. However the final box cannot be inserted into any of the remaining corners.

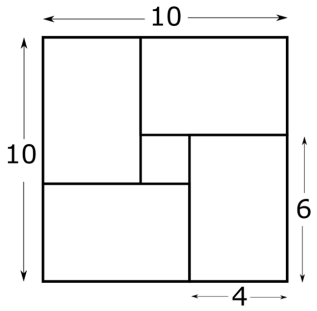

Figure 4: Minimal counter example.

2 Large Neighbourhood Search

Large Neighbourhood Search (LNS) is a single-solution based metaheuristic that aims to find high quality solutions to optimisation problems through iteratively destroying and repairing an incumbent solution. In this exercise, you will build an LNS to the 2D packing problem that utilises the corner heuristic developed in the first section. LNS requires an initial feasible solution to the problem as a starting point. Here we will simply just use the trivial solution where the containers are empty.

2.1 Destroy

Here we partially destroy a solution to our problem by completely unpacking a number of containers. In this way, the boxes in the packed containers re-

main fixed whereas the unpacked containers are relaxed. Clearly, the number of containers that we unpack (the "degree of destruction" parameterised by `fraction_destroyed`) will impact how the search progresses.

Complete the `destroy` function in `lns.py` to partially destroy a given solution by completely unpacking a number of containers. Use random sampling for deciding which containers to destroy (e.g., `random.sample`).

Hint: with the provided framework, any boxes removed from a container will need to be reinserted into the `unpacked` list.

2.2 Repair

We repair a solution by using the corner-heuristic to try to reinsert the unpacked boxes. Recall from section 1.3 that there are many different orderings you could consider providing to the corner heuristic, which you should leverage in your LNS implementation. In particular you will want to introduce enough randomness to get a good balance between exploration and exploitation. Complete the `repair` function in `lns.py` to reconstruct a solution.

2.3 LNS Algorithm

LNS works by iteratively destroying and repairing an incumbent solution. Applications of LNS often differ in how they update the incumbent solution. For example, the simplest version of LNS simply accepts any new solution that has an objective function that is at least as good as the current incumbent. This can be thought of as a hill-climbing method. More advanced methods, such as Simulated-Annealing, allow the possibly for the incumbent solution to be replaced by a solution with a worse objective. Complete the `lns` function in `lns.py` to implement the LNS algorithm using your destroy and repair functions. Design it to accept a new solution if the objective is at least as good as the incumbent solution.

Hint: you'll likely want to "deepcopy" the incumbent solution prior to destruction, so that we have a solution to continue from if that iteration doesn't lead to an improvement.

2.4 Experiments

Evaluate your LNS solver on instances `case-4-100.json`, and `case-5-200.json`. Play around with different parameterisations of the corner heuristic (order-

ings) and the fraction of containers destroyed. What is the best parameterisation combination you find? State the best objective you achieve for each of the two instances after 300 iterations, and plot the corresponding solutions.

Hint: You should be aiming for a solution with an objective above 800 after 300 iterations for `case-4-100.json`.

The best parameterisation I found was preferencing the higher weight boxes first, and then randomly sampling the corners, orientations and containers. The fraction destroyed at 0.5 seemed to work well. This achieve a solutions of 821 and 1771 for the two instances. The solutions are plotted below.

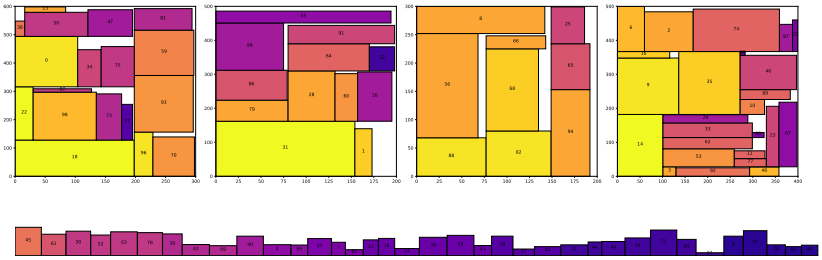

Figure 5: Solution for `case-4-100.json`.

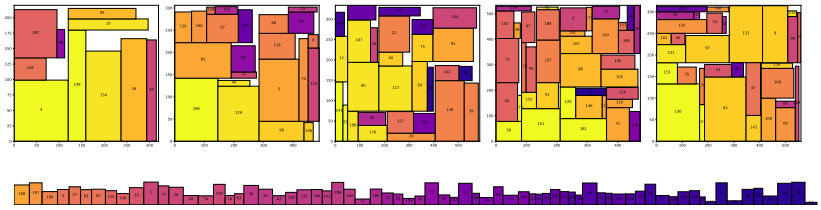

Figure 6: Solution for `case-5-200.json`.

2.5 Benefits?

Re-run your code but with `fraction_destroyed = 1.0` (assuming this wasn't the best you found in the previous section). Explain the implication of this parameter setting. By comparing the solutions after 300 iterations to your

best parameterisation solutions, discuss whether or not this provides evidence that LNS is a beneficial approach for solving this 2D packing problem.

This represents the case where we are simply just resolving the corner heuristic repeatedly. As our parameterisation randomly samples, this can lead to a different solution each time. This is as if LNS isn't playing any role, since we are considering the completely problem as the neighbourhood each time.

With the fraction destroyed at 1.0, we get solutions 802 and 1737 for the two instances over 300 iterations, which is a reduction compared to our best parameterisation. This indicates that the LNS approach of restricting our problem to more localised neighbourhoods can lead to better outcomes as it better focuses the search.

I'd note that this evidence is not particularly strong and would require further experimentation and looking at instances with larger numbers of containers.