VEHICLE LOADING PROBLEM

Edoardo Fadda, Emilio Leonardi, Camilla Massaro

April 4, 2023

ABSTRACT

Supply chain demands are reaching new heights, pushing companies to find new ways to boost efficiency. Loading and unloading are some of the most routine processes in logistics and are an excellent place to start. Tackling truck loading efficiency presents many opportunities to streamline operations and reduce costs.

1 Introduction

The problem objective is to pack a set of items into stacks and pack the stacks into trucks to minimize the number of trucks used.

In terms of data volume, a large instance can contain up to 260000 items and 5000 planned trucks, over a horizon of 7 weeks. Hence, 37142 items per day.

2 Problem Description

Three-dimensional trucks contain stacks packed on the two-dimensional floor. Each stack contains items that are packed one above another. There are three correlated dimensions (length, width, height) and an additional weight dimension. We call horizontal dimensions the length and the width. The position of any element (item or stack) is described in a solution by its position in the three axis X, Y, and Z. There is one origin per truck, corresponding with the leftmost downward position (cf Figure 1).

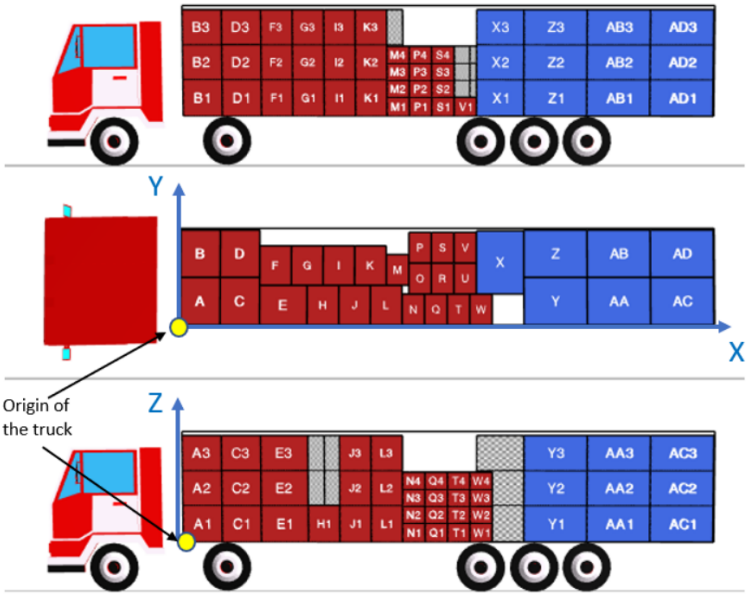


Figure 1: Views from top to bottom: right view, above view, and left view of a truck, seen from the rear of the truck. The truck contains 96 items (A1 to AD3) packed into 30 stacks (A to AD).

2.1 Items

An item i is a packaging containing a given product. It is characterized by:

- length l_i, width w_i, and height h_i (by convention, the length is larger than the width).
- weight ω_i
- stackability code p_i
- maximal stackability P_i
- forced orientation O_i
- nesting height $\bar{h}_i$

The stackability code p_i is used to define which items can be packed into the same stack: only items with the same stackability code can be packed into the same stack. Items with the same stackability code share the same length and width. But the opposite is false: items with identical length/width may not share the same stackability code.

An item i is associated with maximal stackability P_i. It represents the maximal number of items i in a stack.

The items may be rotated only in one dimension (horizontal), i.e. the bottom of an item remains the bottom. An item may be oriented lengthwise (the item's length is parallel with the length of the truck), or widthwise (the item's length is parallel with the width of the truck). In Figure 1, stack X is oriented widthwise, while stack Y is oriented lengthwise. Item i may have a forced orientation O_i, in which case the stack s containing item i must have the same orientation.

An item i has a nesting height $\bar{h}_i$ which is used for the calculation of the height of stacks: if item i_1 is packed above item i_2, then the total height is $h_{i_1} + h_{i_2} - \bar{h}_{i_1}$ (cf Figure 2). For the majority of items, the nesting height is equal to zero.

2.2 Trucks

A vehicle v is characterized by its dimensions length/width/height (L_v, W_v, H_v), its maximum authorized loading weight W_v, and its cost C_v. The truck's cost represents a fixed cost that does not depend on the number of items loaded into the truck.

For structural reasons, there is a maximal total weight TMM_v for all the items packed above the bottom item in any stack loaded into truck v.

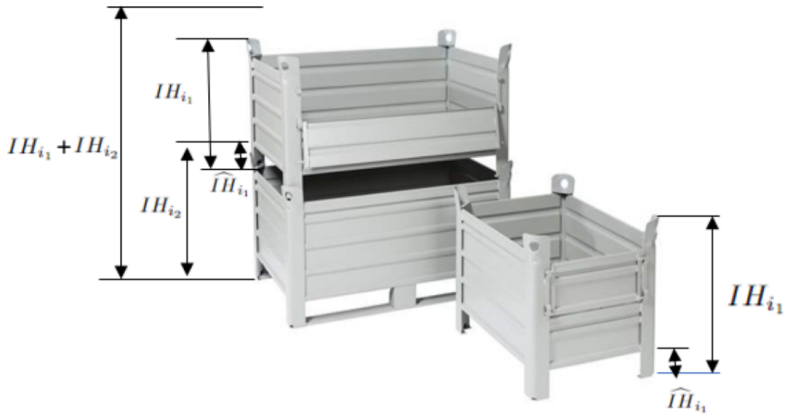


Figure 2: Example of a stack of 2 items with nesting height

2.3 Stacks

While items and planned trucks are parameters for the problem, stacks represent the results of the optimization. A stack contains items, packed one above the other, and is loaded into a truck. A stack s is characterized by its dimensions length/width/height (l_s, w_s, h_s) and its weight ω_s. The length and width of a stack are the length and width of its items (all the items of a stack share the same length/width). A stack's orientation is the orientation of all its items. There is a unique orientation (lengthwise or widthwise) for all the items of a stack. The stack is characterized by the coordinates of its origin (x_s^o, y_s^o, z_s^o) and extremity points (x_s^e, y_s^e, z_s^e).

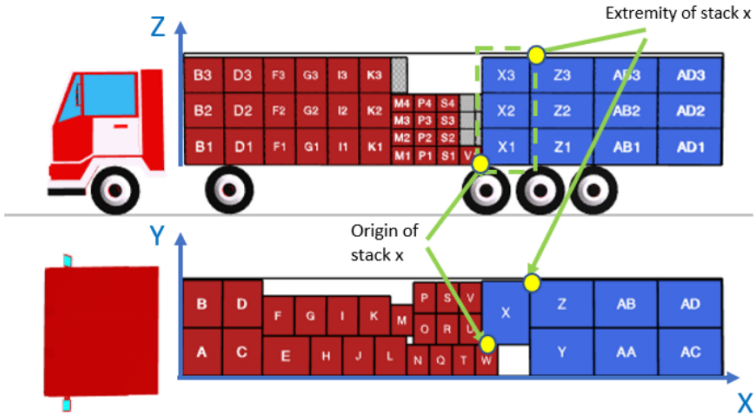


Figure 3: Origin and extremity points of a stack.

As shown in Figure 3, the origin is the leftmost bottom point of the stack (the point nearest the origin of the truck), while the extremity point is the rightmost top point of the stack (the farthest point from the origin of the truck). The coordinates of origin and extremity points of a stack must satisfy the following assumptions:

- $x_s^e - x_s^o = l_s$ and $y_s^e - y_s^o = w_s$ if stack s is oriented lengthwise
- $x_s^e - x_s^o = w_s$ and $y_s^e - y_s^o = l_s$ if stack s is oriented widthwise
- $z_s^o = 0$ and $z_s^e = h_s$

A stack's weight ω_s is the sum of the weights of the items it contains:

$$\omega_s = \sum_{i \in I_s} \omega_i.$$

The stack's height h_s is the sum of the heights of its items, minus their nesting heights (see Fig 2):

$$h_s = \sum_{i \in I_s} h_i - \sum_{i \in I_s, i \neq bottomitem} \bar{h}_i$$

3 Problem

The objective function is the minimization of the total cost, subject to the constraints described above and that no free space between objects: Any stack must be adjacent to another stack on its left on the X axis, or if there is a single stack in the truck, the unique stack must be placed at the front of the truck (adjacent to the truck driver):

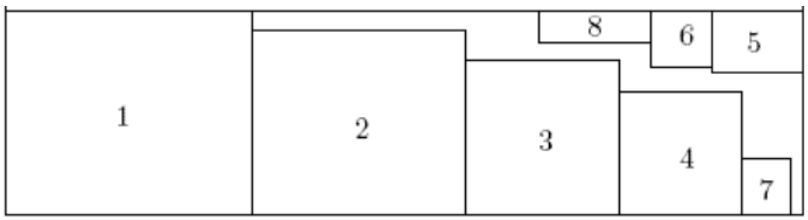


Figure 4: Example of non-feasible packing.

4 Data

4.1 Input Data

Each instance has two files: the file of items and the file of vehicles. An example of file of item is:

id_item	length	width	height	weight	nesting_height	stackability_code	forced_orientation	max_stackability
I0000	570	478	81	720	47	0	w	4
I0001	570	478	96	256	47	0	w	4
I0002	1000	975	577	800	45	1	n	100

NB: the dimension of the items is set lengthwise.

The file of vehicles:

id_truck	length	width	height	max_weight	max_weight_stack	cost	max_density
V0	14500	2400	2800	24000	100000	1410.6	1500
V1	14940	2500	2950	24000	100000	1500	1500

5 Rules

- Group up to 3 people
- Select one heuristic in the provided set, max 1 group for each heuristic framework.
- Write a report on the solution method developed. Each report must have a table with the name of the instance, the cost, and the computational time for all the instances provided.
- you will work on a github project, hence 1 account per group is required.

Your final evaluation will consider:

- Quality of the report. Do not describe the problem. Focus on:
  - literature review (if you used other work)
  - your heuristic
  - computational results

- Quality of the code
- Bug reporting
- Computational results
- Presentation (10 minutes maximum)

6 Heuristics

In this section, we present the heuristics that you are going to select.

1. **Exact model with gurobi** use gurobi to solve the problem. You will be evaluated on a set of small instances to test instances.
2. **Exact model with Or Tool CP-solver** use Or Tool CP-solver to solve the problem. You will be evaluated on a set of small instances to test instances.
3. **Local Solver**: use local solver to solve the problem [1]
4. **Large Neighborhood Search**
5. **GRASP**
6. **Reduced Variable Neighborhood search**
7. **Basic Variable Neighborhood search**
8. **Adaptive Large neighborhood search**
9. **Local Solver based heuristic**: use local solver to solve subproblems
10. **OR tool**: use OR tool to solve subproblems [2]
11. **Genetic Algorithms**
12. **Tabu search** material
13. **Simulated Anealing**
14. **Particle swarm optimization** material
15. **Column Generation**: Use column to solve the problem or the 2D bin packing problem. 3 extra points if you choose this one due to difficulty [1].
16. **Decision Rule** (only for a group of 1 person): ordering the items using a given criterion and then trying to allocate the items considering a bin at the time. When no more items can be allocated in the current bin, we close such a bin and open a new one. The process stops when all items have been allocated. (see Best-fit bin packing, [2], ...)
17. **Kernel search** apply kernel search on the 2D bin packing problem
18. **Positions and Covering** for the 2D bin packing problem (material)
19. **Path-relinking**
20. **Ruin and recreate**
21. **Cross entropy** material (section 3.1 in particular).
22. **Ejection Chains**
23. **Ant colony optimization** [3]
24. **Scatter search** [3]
25. **Relaxation Induced Neighborhood Search** for solving the 2D bin packing problem [3]
26. **Corridor method** for solving the 2D bin packing problem [3]
27. **Fore-and-Back** for solving the 2D bin packing problem [3]
28. **Variable-Depth Methods** [3]
29. **Lagrangean Relaxation**

[1] `https://www.localsolver.com/`

[2] see AddNoOverlap2D `https://developers.google.com/optimization/cp/cp_solver`

30. **Fix and Relax**

Each project must fulfill the requirement, for the rest you can implement it according to your own will.

A list of suggestions that you can choose to implement

- You have three decisions to take:
  - how to create the stack
  - where to locate them
  - 2D bin packing problem

  One possibility is to start from an initial guess and then improve it.
- For the 2 and 3-dimensional bin packing problem see [4]
- Build the stack heuristically, then solve a 2d knapsack problem, and improve the stack decision and the assignment.
- Force an initial solution to the 2d knapsack problem in an exact solver to improve convergence.
- Solve the 2d knapsack problem by fixing several variables.
- Remove stacks from the incumbent solution and solve an exact model for setting them again.

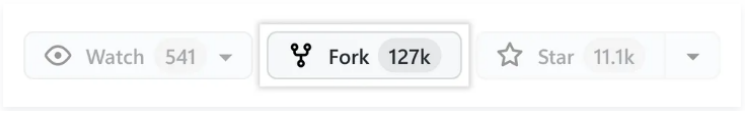
You will all work on the same git repository hence you must work on your repository and strictly follow what we will say in the lecture.

7 Project Management

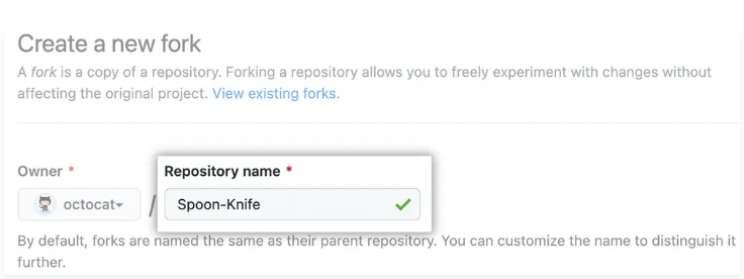
All the groups will work on the same project. Thus, you need to know a little about IT project management. We will use Git a free and open-source distributed version control system.

To work on the project you need to:

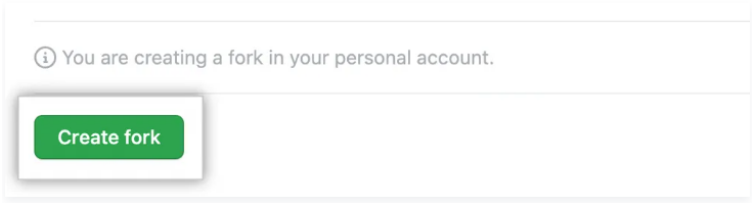
1. Create a fork of the project, first clicking Fork in the main project



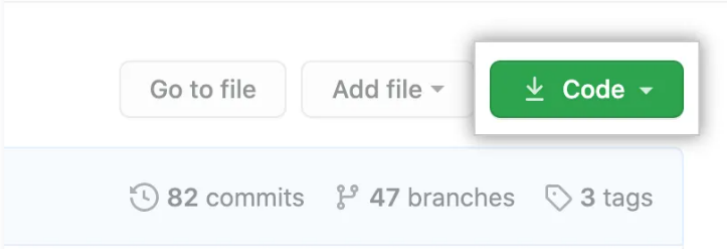
   select the owner of the forked repository and type the repository name, equal to the one of the main repository



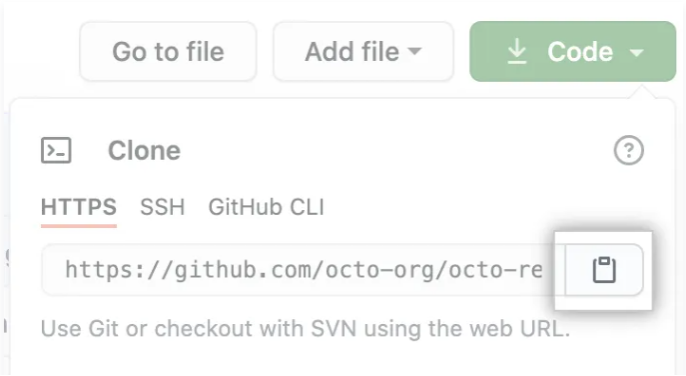
   click on create fork



2. Navigate to your fork and click "Code"



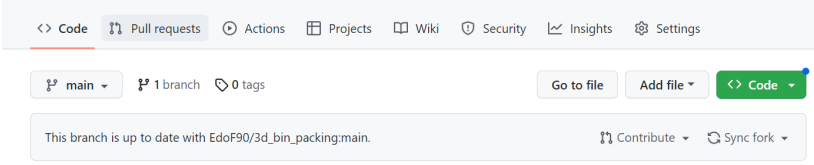
then copy the URL under "HTTPS"



3. Clone the forked project on your PC, opening the Git Bash where you want to put the folder[3]. Type git clone, paste the URL copied before, and press Enter. Something like this will appear:

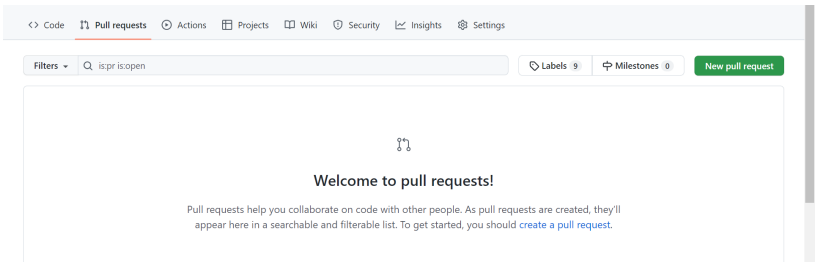
```
$ git clone https://github.com/YOUR-USERNAME/Spoon-Knife
> Cloning into `Spoon-Knife`...
> remote: Counting objects: 10, done.
> remote: Compressing objects: 100% (8/8), done.
> remove: Total 10 (delta 1), reused 10 (delta 1)
> Unpacking objects: 100% (10/10), done.
```

4. Change the code in your local repository as you want on (remaining in your folder)
5. Push the changes on your fork, by typing the following commands:
   (a) git status to understand what you will push
   (b) git add <file name> to add a file, or git add . to add all of your changes
   (c) git commit -m "string describing what you do"
   (d) git push
6. Go on your GitHub and click on pull request

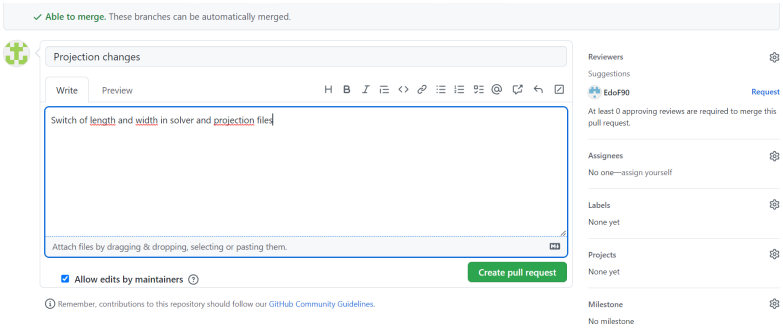


then create new pull request

[3]you can also use softwares as SourceTree, ...



7. Always add a description to let us know what you do, then click on create pull request



When to do a pull request?

- when you create your folder
- when you have reached a codebase state you want to remember and you want us to check. Prefer to do several small pull requests instead of just a big one. In this way it is easier for us to check.

When to commit?

Think of coding as rock climbing in this context. You climb a bit, and then you put an anchor in the rock. Should you ever fall, the last anchor you planted is the point that secures you, so you'll never fall more than a few meters. Same with source control; you code a bit, and when you reach a somewhat stable position, you commit a revision. Should you ever fail horribly, you can always go back to that last revision, and you know it's stable.

If you work on a team, it's customary to make sure whatever you commit is complete, makes sense, builds cleanly, and doesn't break anyone else's stuff. If you need to make larger changes that might interfere with other peoples' work, make a branch so you can commit without disturbing anyone else.

References

[1] David Pisinger and Mikkel Sigurd. The two-dimensional bin packing problem with variable bin sizes and costs. *Discrete Optimization*, 2(2):154–167, June 2005.

[2] MarcoA. Boschetti and Aristide Mingozzi. The two-dimensional finite bin packing problem. part II: New lower and upper bounds. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2), June 2003.

[3] Vittorio Maniezzo, Marco Antonio Boschetti, and Thomas Stützle. *Matheuristics: Algorithms and implementations*. Springer Nature, Cham, Switzerland, April 2021.

[4] José Fernando Gonçalves and Mauricio G.C. Resende. A biased random key genetic algorithm for 2d and 3d bin packing problems. *International Journal of Production Economics*, 145(2):500–510, October 2013.