1.1. Formulating the optimization problem

To represent the assignment of each product i to a container j, we can in analogy to routing and resource allocation problems formulate a matrix $X \in R^{P \times N}$, with $x_{ij} \in \{0, 1\}$ where x_{ij} is either 0 if product i is not assigned to container j and 1 if it is. PS: note that, here, we are using uppercase X to represent a matrix whose elements are represented by x_{ij}.

To formulate the objective function, we introduce below an auxiliary variable $\boldsymbol{y}$, where $\boldsymbol{y}$ is a vector of size N, $y_j \in \{0, 1\}$, $\forall j \in \{1, \dots, N\}$ and

$$y_j = \begin{cases} 1, & if \sum_{i=1}^{P} x_{ij} > 0 \\ 0, & otherwise \end{cases}$$

i.e. it is 0 if container j is not used or 1 if container j is being used. Using the auxiliary variable, we can formulate in the following an objective function which counts the number of containers $f(\boldsymbol{y})$ as follows:

$$f(\boldsymbol{y}) = \sum_{i=1}^{N} y_i$$

Having formulated design variables and objective function, we can move to define the constraints. The first constraint enforces that the number of used containers is

$$f(\boldsymbol{y}) \leq N$$

i.e. at most N containers should be used. In principle, this constraint can be considered to be fulfilled by design (i.e., this can be an implicit constraint), as at most N containers can be used in our design variable X.

The next set of constraints are explicit constraints to ensure that each product is assigned to one and only one container. Particularly, we want to enforce for each product:

$$\sum_{j=1}^{N} x_{ij} = 1, \qquad \forall i \in \{1, \dots, P\}.$$

which can be rewritten as:

$$h_i(X) = \left(\sum_{j=1}^{N} x_{ij} \right) - 1 = 0, \qquad \forall i \in \{1, \dots, P\}.$$

The summation in the equation above calculates, for each product $i \in \{1, \dots, P\}$, the number of times that this product is allocated to a container. This is because x_{ij} can only take values 0 (if the product i is not loaded into container j) or 1 (if it is loaded). Therefore, this summation will take value 1 if product i is allocated to one and only one container, and a value different from 1 otherwise. Once we subtract 1 from this summation, h_i will get the value zero if product i is allocated to one and only one container, and a value different from zero otherwise.

The final set of constraints ensures that no container exceeds the maximum volume V_{max}:

$$\sum_{i=1}^{P} v_i x_{ij} \leq V_{max}, \qquad \forall j \in \{1, \dots, N\}.$$

which can be rewritten as:

$$g_j(X) = \sum_{i=1}^{P} v_i x_{ij} - V_{max} \leq 0, \quad \forall j \in \{1, \dots, N\}.$$

Meaning that for a given container j the sum of the volumes v_i for each product i that is allocated to that container should not exceed the maximum number of available container space V_{max}.

In principle, we now have all the ingredients together to start tackling the optimization problem. In the following, we will therefore design an optimization algorithm to solve it.

1.2. Designing a simple algorithm for solving it

In the following, we will use the Simulated Annealing algorithm from Lecture 9.1 & 9.2 as a base to design an algorithm for solving our optimization problem. Recall, that the Simulated Annealing algorithm consists basically of two parts, with the first one being the random initialization of a start solution, and the second one being a loop which is repeatedly evaluated such that iteratively more optimal solutions are generated. We also need to decide on a representation and on the strategy to deal with the constraints.

- **Representation:** a direct representation of the design variable can be used. Note: this is not necessarily the best possible representation to be used, despite being a representation that works. Can you think of other possible representations?
- **Initialization:** to initialize the Simulated Annealing algorithm, we draw randomly a product from the set of all available products, remove it from the set, and test whether we can fit into the first container. If the first container still has available capacity, we assign the product to it. If not, we move to the next container. We successively try to fit the product into each container until we have found one which still has available capacity. PS: Note that there will always be at least one container that can accommodate the product so long as each product's volume does not exceed V_{max}. Why is that the case?

  The pseudocode is outlined as follows:

  - **Set $X \in R^{P \times N}$ to the initial solution with $x_{ij} = 0 \;\; \forall (i,j) \in \{1, \cdots, P\} \times \{1, \cdots, N\}$**
  - **Set $S = \{1, \cdots, P\}$ to the set of all available products**
  - **While $S \neq \emptyset$:**

- Draw one product $m \in S$ randomly and remove it from the set such that $S = S \setminus \{m\}$.
  - Set $n = 1$
  - While $n \leq N$:
    - If $\Sigma_{i=1}^{P} v_i x_{in} + v_m \leq V_{max}$
    - Then
      - $x_{mn} = 1$
      - BREAK
    - Else $n = n + 1$
- ## Terminate the for loop as soon as $S == \emptyset$

- **Neighbourhood operator:** To generate a solution in the neighbourhood of X, we choose a random product, and generate new solutions, by first making a copy of X and reassigning the product to a random new container. The pseudocode is outlined as follows:

  - Given $\mathbf{X} \in \mathbf{R}^{\mathbf{P}\times\mathbf{N}}$ to be a previously generated solution
  - $\mathbf{m} = \mathbf{randomInt}(\mathbf{1}, \mathbf{P})$ // this function will pick a product uniformly at random between 1 and P (inclusive).
  - $\mathbf{S} = \{\mathbf{1}, \ldots, \mathbf{N}\}$ // Set of all available containers
  - Repeat:
    - $\mathbf{X}' = \mathbf{clone}(\mathbf{X})$ // this function will create a new variable that is a clone (a full copy) of X.
    - $\mathbf{k} = \mathbf{argmax}_{\mathbf{j}} (\mathbf{x'_{mj}})$ // this function will return the index of the container j that contains the product m.
    - $\mathbf{S} = \mathbf{S} \setminus \mathbf{k}$
    - $\mathbf{n} = \mathbf{randomInt}(\mathbf{S})$ // pick a container uniformly at random between 1 and N (inclusive), except for the current container k
    - $\mathbf{S} = \mathbf{S} \setminus \mathbf{n}$
    - $\mathbf{x'_{mk}} = \mathbf{0}$ // remove product m from container k
    - $\mathbf{x'_{mn}} = \mathbf{1}$ // add product m to container n
  - Until $\Sigma_{i=1}^{P} v_i x_{in} \leq V_{max}$ or $\mathbf{S} == \emptyset$ // this termination condition will check whether the maximum volume constraint is satisfied for the container n into which we moved product m
  - return X' as new solution if $\Sigma_{i=1}^{P} v_i x_{in} \leq V_{max}$ or null otherwise

  Note that the pseudocode above could return null for a given solution X that may actually have some possible neighbour that satisfies the constraints. How could you improve on that?

- **Constraint handling:** the design variable ensures that at most *N=P* containters are used. The initialization operator above will always ensure that every product is allocated to one and only one container, and that the maximum volume of the container is not exceeded. After that, the neighbourhood operator above will always ensure that the randomly picked product is moved to a single new container that can accommodate it, also satisfying these two constraints.

1.3. Assumptions:

- We are assuming that the volume of each individual product does not exceed V_{max}.
- Within this proposed solution, we assumed that each container has the same capacity as well as there are no weight constraints acting upon them. Further, we

have also previously assumed that by there are not any constraints stemming from the shapes of the products.

1.4. Optional Challenge:

- Our objective function does not consider the fullness of the containers. We could potentially modify it to try and make each container as full as possible. Similar to penalties functions that attempt to deal with constraints by guiding the algorithm towards feasibility, a penalty function to penalize containers based on their fullness could potentially provide additional guidance for the algorithm to find solutions that use less containers.

Note: The problem featured within this exercise is commonly known as bin packing problem within the literature.