POLYTECHNIC UNIVERSITY OF TURIN

Master's Degree in Mathematical Engineering

Master's Degree Thesis

Leveraging the Structure of the Classical Assignment Problem to Solve a Large-Scale Production Optimization Problem

Supervisors:

Prof. Paolo BRANDIMARTE
Dr. Francesco ROMITO
Dr. Daniel FACCINI
Silvia COLASANTE
Marco MORETTO

Candidate:

Diego MORICHELLI

November 2024

Contents

Acknowledgments

I would like to express my sincere gratitude to Professor Paolo Brandimarte for making this thesis possible.

I am deeply grateful to my supervisors at Spindox, Francesco Romito, Daniel Faccini, Silvia Colasante and Marco Moretto, for the time they devoted to guiding and helping me at every stage of this work. Their expertise and patience were instrumental in the development of this thesis.

I would also like to express my heartfelt gratitude to my brother, my mother and my father for their support, encouragement, and belief in me. Their constant presence and understanding have been a source of strength and motivation throughout this journey. This accomplishment would not have been possible without them.

Abstract

This thesis explores a complex production and distribution problem that involves various production plants, orders, product types and time-based restrictions.
The goal is to develop an efficient assignment model that optimally allocates production orders to plants while balancing cost, resource limitations, and production capacities.
The problem is formulated as a large-scale assignment challenge, where orders must be assigned to plants capable of meeting demand either on time or within an acceptable delay period.
The primary challenges involve handling complex constraints while minimizing penalties associated with unmet demand, delayed orders, and plant-specific product-type limitations.
To solve this problem, we leveraged the Hungarian algorithm, alongside strategies for clustering orders and modeling the instance in a way that fits the classical assignment problem structure.
This method enabled us to perform a fine-grained assignment of orders to plants.
Orders are clustered based on delivery timelines and capacity needs, and the Hungarian algorithm assigns these clusters to plants in sequence.
Initially, we focused on addressing constraints related to capacities, delays, and unfulfilled orders, as these are the most compatible with the classical assignment problem framework. However, we ultimately succeeded in incorporating a more complex type of constraint.
Complexity penalties were integrated to account for plant-specific product type limitations, yielding positive results and resulting in a flexible and realistic model.
Experimental evaluations indicate that this method effectively minimizes computational demands while achieving cost-effective solutions.
The implemented sequential batching approach improves scalability, while the complexity penalties ensure that plant-specific constraints are effectively integrated into the solution.
Although adding constraints would typically shift the problem towards a generalized assignment problem, we managed to structure the data in a way that allowed us to formulate it as a classical assignment problem, solvable using the Hungarian algorithm.
This research advances the field by presenting a scalable and practical solution to complex production-distribution assignment problems, with direct applications in industries that require adaptable and cost-effective production planning.
The solutions obtained can also serve as a valuable starting point for industries to apply further refinement and improvement techniques.

Literature Review

The foundation of this thesis lies in the assignment problem, an extensively studied topic within operations research.
Early contributions define the problem as finding the minimum-cost assignment of jobs to workers, where each assignment incurs a fixed cost .
The Hungarian algorithm, early formulated in [1], provides an optimal solution to the assignment problem by iteratively minimizing rows and columns of the cost matrix in polynomial time.
Subsequent research,like the article [2],includes the extension of this method to handle rectangular matrices, making it suitable for complex, large-scale applications with unequal rows and columns.
Our study incorporates advancements in data modeling and cost computation techniques to develop an assignment problem framework capable of managing complex constraints.
These contributions align with recent literature on mixed-integer programming and combinatorial optimization for production-distribution systems, emphasizing the scalability and efficiency of algorithmic adjustments like sequential batching and penalty functions for delay handling.

1 Introduction

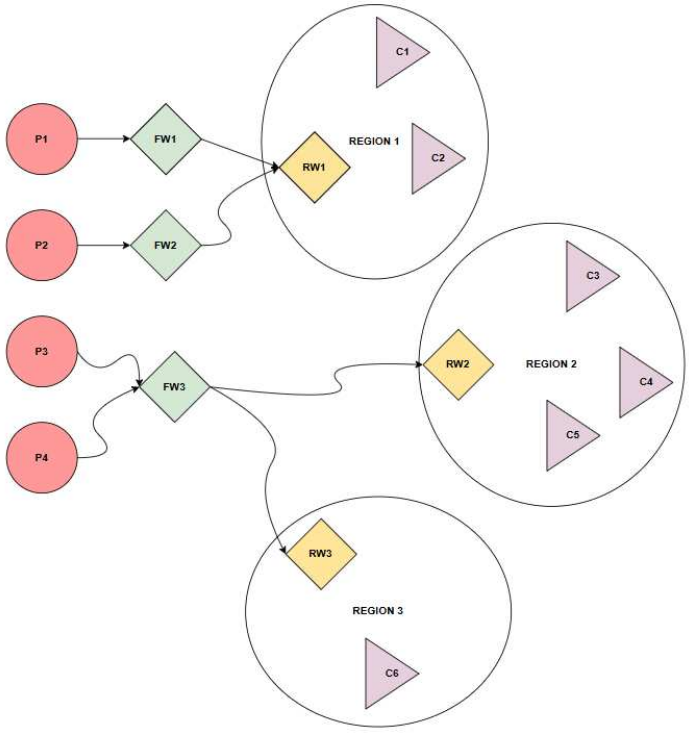

Figure 1: Network scheme

We are addressing a production and distribution problem in which various goods must be produced and delivered to different clients. We are provided with a list of orders, each specifying the requested product type, the expected month of delivery, and the facilities capable of producing the order.
Additionally, we have access to information about the production plants, each of which is characterized by unique constraints.

Figure 1 shows the network we just described. A detailed description follows:

- **Production plants** are the factories that produce the requested goods.
- **Factory warehouses** are represented by the index **FW**.They store the goods produced before they are shipped.
- **Regional warehouses** represented with **RW** indicate the warehouses located in different regions. Regional warehouses serve all the clients of their respective region

We can decompose the problem, treating Production and Distribution separately.

1.1 Production Phase

We have a set of nineteen plants where we can produce different kind of products.
The limits we are facing are the following:

- Plants have a limited capacity after which they can exploit their extra capacity at an higher production cost.
- Not all plants have the technology needed to produce every product type.

  We say a product is compatible with a plant if they can be produced in it.
- Each plant has a limited amount of different product types.
- Plants have, for each month, a limited degree of newly introduced product types. We will say that they present a limited phase in.

1.2 Distribution Phase

The produced products have to be distributed to clients located in different locations.

- Each plant has its own factory warehouse.
- Each client is associated to a location, where there is a regional warehouse.
- Each route going from a given factory warehouse to a regional one has its own transportation time and cost.
- Some orders require direct shipping from factory warehouses.
- Orders sometimes allow a delay.

1.3 Scale of the instance

The problem is large-scale primarily due to the high cardinality of product types.
While the number of plants and orders is in the tens and hundreds, there are thousands of products to manage and assign.
This, combined with complex constraints, makes any attempt to use classic solvers extremely inefficient.
The dimension of the demand are the following:

- We need to assign a total of 66,024 orders.
- The total amount of goods to be produced is 70,302,284 units.
- The orders involve 3,782 different product types.
- The total production capacity amounts to 124,635,394 units.
- There are 19 production plants.
- Orders are distributed over an 18-month time horizon.

These numbers make the problem infeasible for classical solvers, as their exponential complexity would result in an extremely long resolution time.

2 Focus of the Research

Advised by professionals that are trying to tackle the problem, we focused on the production phase, as it includes the most challenging optimization part and comprehends the biggest costs in action.
Another reason for which we do it is that the production optimization phase results in a large scale integer programming optimization problem while the transportation phase, due to its similarity with more famous problems like the shortest path, can be solved more efficiently by simpler algorithms.
Therefore, we focused on assigning orders to the appropriate plants to minimize delayed and unfulfilled orders while simultaneously adhering to plant-specific constraints. .

2.1 Simplifying hypothesis

In our initial approach to tackling the problem, we made some simplifications:

- **Transportation times and costs**:
  We do not consider any of them, instead we work on a simplified environment where once an order or a part of it is assigned to a plant, it is considered as immediately delivered.
- **Delay Cost**:
  Instead of assigning a penalty related to the cumulative amount of goods in delay, we assign at each individual order a delay penalty cost which is proportional to the entity of the delay.
  Therefore, if we produce an order with a delay of $\mathbf{x}$ months, we will face a penalty equal to $\mathbf{f}(x)$, with $\mathbf{f}$ strictly increasing in x .
- **Plant Maximum Complexity**:
  In the initial experiments, we disregarded these constraints.
  Later, we incorporated them, initially treating them as a soft constraint and subsequently as a hard one.
- **Phase-ins**:
  We ignored this constraint.
- **Costs**:
  The production of any product incurs costs based on principles that are consistent across all product types.

2.2 Mathematical model

Our goal is to find a solution that minimizes both the total number of unfulfilled orders and the delays.

2.2.1 Variables legend

- i : production plant i.
- j : order j.
- k : product type k.
- $k(j)$: product type of the order j.
- y : month y.
- $y(j)$: month in which order j is expected to be produced.
- z_i^{ky} : quantity of product k produced in time y in plant i.
- z_i^y : total amount of goods produced at time y by plant i.
- $b_j^{y'}$: portion of order j produced with delay at time y'.
- v_i^y : quantity of products produced by plant i at time y using extra capacity.
- m_j : portion of order j not evaded.

2.2.2 Hyper-parameters legend

- $\mathbf{D}_j$:

  Amount of order j.
- $\mathbf{C}_i^y$:

  default capacity of plant i at time y.
- $\mathbf{EX}_i^y$:

  extra capacity of plant i at time y.
- **CostProd** :

  unitary cost for producing each good using default capacity.
- **CostSurplus** :

  unitary cost of production using extra capacity.
- **CostDelay**$(y(j), y')$:

  unitary penalty incurred producing goods requested by client j with delay.
  $y(j)$ indicates the expected production month for order j.
  y' is a month that falls within the allowable delay period for the specific order.
- **CostInevasion** :

  unitary cost associated with the non-fulfillment of an order.

2.2.3 Costs hierarchies

Different types of costs vary significantly in magnitude, providing a clear indication of which aspects to prioritize during optimization:

$$\textbf{CostSurplus} << \textbf{CostDelay} << \textbf{CostInevasion}.$$

The worst-case scenario is not evading orders by a significant margin.
Furthermore, it is preferable to utilize any extra capacity available at a plant rather than resorting to delayed order fulfillment.

It may be noted that, when introducing the hyper-parameter $\textbf{CostDelay}(y(j), y')$, we did not specify that y' must be a delayed production month permitted by order j.
This is not an issue for our model formulation, as we can set $\textbf{CostDelay}(y(j), \tilde{y}) = \textbf{CostInevasion}$ in cases where $\tilde{y}$ is not an allowed production month for the order.

2.2.4 Optimization Model

$$\min_{z,b,v,m} \sum_{i,y} \mathbf{CostProd}\, z_i^y + \sum_{i,y} \mathbf{CostSurplus}\, v_i^y + \sum_{j,y'} \mathbf{CostDelay}(y(j), y')\, b_j^{y'} + \sum_{j} \mathbf{CostInevasion}\, m_j$$

$$\text{s.t.} \quad z_i^y \leq \mathbf{C}_i^y + \mathbf{EX}_i^y \qquad \forall\, i, y$$

$$v_i^y \geq z_i^y - \mathbf{C}_i^y \qquad \forall\, i, y$$

$$z_i^y = \sum_k z_i^{ky} \qquad \forall\, i, y$$

$$b_j^{y'} \leq \sum_i z_i^{y'} \qquad \forall\, j, y'$$

$$\mathbf{D}_j \leq m_j + \sum_{y(j)<y'} b_j^{y'} + \sum_i z_i^{y(j)} \qquad \forall\, j$$

$$z_i^{ky} \geq 0 \qquad \forall\, i, k, y$$

$$b_j^{ky'} \geq 0 \qquad \forall\, j, k, y'$$

$$m_j \geq 0 \qquad \forall\, j$$

- **Constraint 1**
  The amount produced in plant i at time y cannot exceed the total capacity of that plant in that month which is the ordinary capacity plus the extra capacity.

- **Constraint 2**
  The quantity v_i^y defines the variable related to the quantity of products produced using extra capacity in plant i at month y.

- **Constraint 3**
  The total amount of goods produced by plant i at time y is the sum of all the amounts of different products produced.

- **Constraint 4**
  The portion of order j we decide to produce at any time y' beyond the expected time $y(j)$ will increase the future requested production.

- **Constraint 5**
  The quantity specified in order j, denoted as D_j, is divided into three parts: one produced on time, one produced with a delay, and a portion that may not be produced.

- **Constraints 6 7 and 8**
  These constraints are purely mathematical, ensuring that all included variables are non-negative

3 Structure of the Data

We began our work with two DataFrames: one containing details of the orders and the other with information about the plants.

3.1 Plants DataFrame

The DataFrame contains the following columns:

- **plant_id**: Identifies each production plant.
- **month**: Indicates the month associated with the data entry.
- **capacity**: The default production capacity of the plant for the given month.
- **extra_capacity**: The additional capacity available beyond the normal production capacity.
- **max_product_types**: The maximum number of different product types that can be produced at the plant.

	plant_id	month	capacity	extra_capacity	max_product_types
0	0	1	20479	2048	41
1	1	1	292185	29219	80
2	2	1	430222	43022	126
3	3	1	1261253	126125	238
4	4	1	37219	3722	2147483647
...	...	...	...	...	...
337	14	18	336192	33619	73
338	15	18	693700	69370	167
339	16	18	96800	9680	62
340	17	18	477972	47797	92
341	18	18	9658	966	2147483647

Figure 2: Plants DataFrame

Figure 2 shows the essential features of the plants we are working with.
As previously mentioned, we have nineteen plants across an eighteen-month time horizon, resulting in a dataframe with 342 entries.

3.2 Orders Dataframe

	product_type	prod_month	quantity	in time	in delay
10403	162	4	10000	[[2, 1], [2, 2], [2, 3], [2, 4], [6, 1], [6, 2...	[[2, 5], [2, 6], [6, 5], [6, 6], [11, 5], [11,...
19012	162	6	10000	[[2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6...	[[2, 7], [2, 8], [6, 7], [6, 8], [11, 7], [11,...
21642	162	7	10000	[[2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6...	[[2, 8], [2, 9], [6, 8], [6, 9], [11, 8], [11,...
26795	162	8	10000	[[2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6...	[[2, 9], [2, 10], [6, 9], [6, 10], [11, 9], [1...
26796	162	8	10000	[[2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6...	[[2, 9], [2, 10], [6, 9], [6, 10], [11, 9], [1...
...	...	...	...	...	...

Figure 3: Orders DataFrame

The DataFrame containing the details of the orders contain the following columns:

- **product_type**: Identifies each production plant.
- **month**: Indicates the month associated with the data entry.
- **prod_month**: The month in which the order is expected.
- **in time**: This set of tuples contains all possible plants that can produce the given order.
  The first element in each tuple represents the plant, while the second represents the month.
- **in delay**: This entry has a similar structure to **in time**, with tuples containing plant and month information.
  However, in this case, the months in the tuples extend beyond the expected month for the order.

4 0-1 Assignment problem

4.1 Assignment problem state of the art

The assignment problem is a classic operations research problem where n jobs have to be assigned to n workers at the minimum cost possible.
Each worker has to have one and only one job assigned. The costs can be summarized with a $n \times n$ cost matrix where each C_{ij} entry contains the cost of assigning the job $i \in \{1, \dots, n\}$ to the worker $j \in \{1, \dots, n\}$.

	worker 1	**worker 2**	**.**	**.**	**.**	**worker** n
job 1	C_{11}	C_{12}	.	.	.	C_{1n}
job 2	C_{21}	C_{22}	.	.	.	C_{2n}
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
job n	.	.	.	.	.	C_{nn}

4.1.1 Model formulation

As discussed in [1] and in most of operations research books, the classical assignment problem mathematical model is

$$\min_x \sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} x_{ij}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1 \qquad i \in \{1, \dots, n\}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j \in \{1, \dots, n\}$$

$$x_{ij} \in \{0, 1\} \qquad i \in \{1, \dots, n\},\ j \in \{1, \dots, n\}.$$

This is a combinatorial problem therefore, without using an heuristic, it is solved by exponential time algorithms.
This may be problematic for us because the scale of our instance is very large.
Luckily an exact resolution method exists.

4.2 Hungarian algorithm

The significance of the Hungarian algorithm lies in its ability to find the optimal solution in polynomial time, making it suitable for large-scale problems. By systematically reducing the matrix and identifying zero elements, the Hungarian algorithm constructs an optimal assignment without having to exhaustively search through all possible combinations, which would be computationally prohibitive for large matrices.

This iterative algorithm guarantees to find the optimal solution in polynomial time.
The first formulation runs in $\mathcal{O}(n^4)$ (see [1], Section II). The algorithm was then improved and brought to have $\mathcal{O}(n^3)$ complexity (see [2]).

4.2.1 Pseudo-algorithm

- **Step 1** Identify the minimum element in each row and subtract it from each element of that row.
- **Step 2** Identify the minimum element in each column and subtract it from each element of that column.
- **Step 3** Draw the least possible number of horizontal and vertical lines to cover all the zeros. If the number of lines equals the order of the cost matrix then go to step 5; otherwise, go to step 4.
- **Step 4** Identify the smallest element among the uncovered elements left after drawing the horizontal and vertical lines in step 3. Subtract this element from all the uncovered elements and add the same element to the elements lying at the intersections of the horizontal and vertical lines. Then go to step 3.
- **Step 5** For each row or column with a single zero, box that zero as an assigned cell. For every zero that becomes assigned, cross out all other zeros in the same column or row. If for a row or a column, there are two or more zeros then choose one cell arbitrarily for assignment. The process is to be continued until every zero is either assigned with box or crossed out. Cells having a box indicate optimal assignment. In case a zero cell is chosen arbitrarily then there may be alternate optimal solution.

We want to leverage the Hungarian algorithm to solve the problem. In order to do so, we need to model our problem properly.

4.3 Generalized Assignment Problem

The classical assignment problem framework includes only one primary constraint: each job must be assigned to a unique worker.
The main limitation of this setup is that it does not allow multiple jobs to be assigned to the same worker, nor does it account for constraints related to worker capacities or the scale of the jobs.
Given that our problem involves various constraints, it might initially seem that it should be modeled and solved as a generalized assignment problem.
However, the challenge with this approach is that no exact resolution method with polynomial complexity exists for the generalized assignment problem, unlike the classical assignment problem.
To address this, we need to find a way to model our problem and its constraints within the pure assignment problem framework, enabling us to solve it using the Hungarian algorithm.

5 Modeling the instance

To apply the Hungarian algorithm effectively, we must model our data into the classical assignment framework.
The classical assignment problem formulation specifies that:

- Each row represents an element that must be assigned to one and only one column.
- Each column corresponds to a worker that can process at most one element.

These constraints prevent us from treating each order individually.
As a result, we needed to model our orders and plants accordingly.
The key requirement is to construct a cost matrix that ensures a one-to-one relationship between row and column elements.

5.1 Order Clustering

We grouped all orders into clusters that meet the following criteria:

- Each cluster contains orders having the same expected delivery month
- Each cluster contains orders with the same **in time** and **in delay** entries.
  This approach allows us to assign the same costs to each order within a cluster.
- Each cluster contains orders that collectively sum to the same total quantity.

By applying this clustering method, we transform individual orders, which vary in terms of expected delivery months and production options, into uniform, clustered elements.

5.2 Splitting the Plants

After clustering the orders, we needed to address the structure of the plants.
To allow multiple elements along the rows to be assigned to the same plant while maintaining the one-to-one structure of the assignment problem, we divided each plant into elements with equal capacities.
In this way, a single plant is represented by multiple elements, each with the same capacity, enabling flexible assignments within the constraints of the algorithm.

5.3 Splitting the Clusters

After splitting the plants into elements with equal capacity, we applied a similar operation to the clusters of orders.
This step is necessary to ensure that each row element can be fully assigned to a column element.

6 Building the Cost Matrix

After performing all the steps defined in 5 we can build the cost matrix.

- Each row element has an associated set of plants that can produce it on time and another set that can produce it with a delay.
- Each column element corresponds to a fraction of a plant's capacity available in a specific month.

For each row, we can iterate over the columns and assign a value to each entry that represents the cost of assigning production of that specific row element to the plant and month associated with the column.

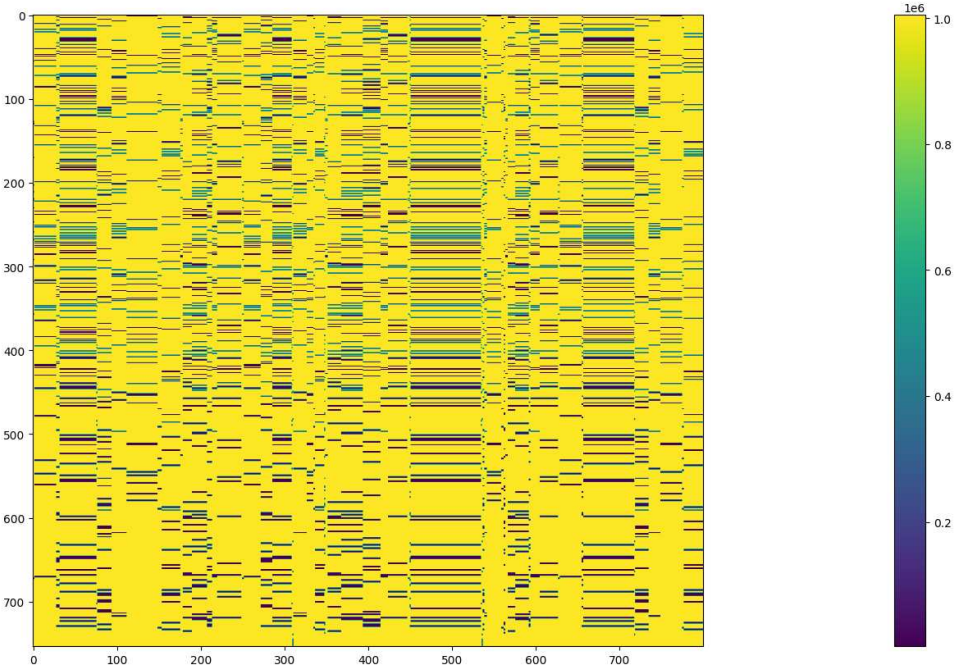

Figure 4: Cost matrix

6.1 Costs

The value $\mathbf{C}_{ij}$ assigned to entry (i, j) of the cost matrix must take into account several factors:

- Whether the production of the element in row i can be completed in the plant and during the month associated with column j.
- The delay incurred if the production of the element is assigned to the month associated with column j.
- The type of capacity referred to by column j.
  If it corresponds to extra capacity, the cost is higher.

We set the different cost values $\mathbf{C}_{ij}$, taking into account all these variables.
The cost can be decomposed as

$$\mathbf{C}_{ij} = \mathbf{Cp}(j) + \mathbf{Ct}(i, j)$$

- $\mathbf{Cp}(j)$ represents the baseline production cost, which depends on the type of capacity associated with column j.
  Columns associated with extra capacity have a higher baseline cost than those associated with default capacity.
- $\mathbf{Ct}(i, j)$ denotes the time penalty incurred by assigning the production of element i, with its expected fulfillment time, to the month associated with column j.

6.2 Handling Incompatibilities

If column j is associated with a plant and month where element i cannot be produced, we set the cost $\mathbf{C}_{ij}$ to be several orders of magnitude higher than all other costs.
Doing so not only effectively represents the high penalty the company would incur for failing to fulfill orders, but it also compels the algorithm to avoid this outcome at all costs.

Figure 4 shows the cost matrix obtained by following the steps outlined above.
We observe that most of the matrix is colored yellow, indicating that a majority of the entries represent incompatibilities.

Following these steps we go from the classical formulation

	Plant 1	**Plant 2**	**.**	**.**	**.**	**Plant** n	**TOTAL**
Order 1	c_{11}	c_{12}	.	.	.	c_{1n}	D_1
Order 2	c_{21}	c_{22}	.	.	.	c_{2n}	D_2
.	.	.	.	.	c_{ij}	.	
Order m	.	.	.	.	.	c_{mn}	D_m
CAPACITIES	C_1	.	.	.	.	C_n	

To this structure:

	Plant 1				Plant 2			Plant 3			.	.
	col 1	col 2	col 3	col 4	col 5	col 6	col 7	col 8	col 9	col 10	.	.
element 1	.	.	.	.	.	.	.	.	.	.	.	.
element 2	.	.	.	.	.	.	.	.	.	.	.	.
element 3	.	.	.	.	.	.	.	.	.	.	.	.
element 4	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
element M	.	.	.	.	.	.	.	.	.	.	.	.

6.3 Cluster Size

While clustering the orders, we must choose the cluster size.
This choice significantly impacts the final dimensions of the cost matrix, as it affects both the number of rows and columns.
From now on we will refer to the size of the clusters as batch size.
In 1 we show how the batch size affects the execution time and the memory needed to process the cost matrix.

6.4 Hungarian Algorithm and Scipy

To perform the classic Hungarian algorithm, the matrix has to be square.
The cost matrices built during the experiments were $\mathbb{R}^{m \times n}$ rectangular matrices with $m \ll n$.

The state of the art in assignment problem theory suggests that, to handle this irregularity, one should add dummy rows to make the matrix square.

The main challenge with this approach is that the matrix dimensions often become too large for memory to handle.
Fortunately, a variant of the algorithm found in the Scipy library addresses this issue. The documentation can be found in [4].
The algorithm performs a variant of the Hungarian algorithm that guarantees to find the optimal assignment to rectangular cost matrices.

6.5 Testing the Algorithm Efficiency

To verify the effectiveness of this variant, we conducted a check.
We transformed the matrix into a square form and applied the classic Hungarian algorithm, then compared the results with those obtained from the original rectangular matrix.
The solution obtained were exactly the same.
In terms of speed, this method—from data modeling to solving the assignment problem—required only a short amount of time, ranging from a few seconds to a minute.

7 Experiments

7.1 Testing the dimensions and the performances of our strategy

First of all we tested how the model behaves using different batch sizes.
As anticipated, using smaller batch sizes leads to a more fine clustering at the price of a larger computation time, both in the creation of the compatibility matrix and in the solving of the Hungarian algorithm.

Batch size	**Matrix Rows**	**Matrix Columns**	**Creation Time [sec]**	**Optimization Time [sec]**
100000	341	975	0.8	0.004
75000	522	1353	0.13	0.008
50000	927	2144	0.257	0.021
25000	2057	4668	0.6	0.1
10000	5719	12094	2.05	0.82
7500	7192	16234	2.42	1.3
5000	9888	24508	3.5	2.48
2500	15994	49483	6.24	11.8

Table 1: Sizes and computational times for different batch sizes

We can see how the cost matrix size grows when we try to decrease the size of the batches.
The hungarian algorithm behaves well, as it keeps running in seconds.
Another factor we took in account is the memory needed for the allocation of the cost matrix:

Batch size	**Matrix Rows**	**Matrix Columns**	**Memory usage [MB]**
100000	341	975	315.07
75000	522	1353	458.6
50000	927	2144	601.85
25000	2057	4668	772.125
10000	5719	12094	1146.22
7500	7192	16234	1476.277
5000	9888	24508	2106.94
2500	15994	49483	3531.25

Table 2: Sizes and memory usage for different batch sizes

Although processing time increases with more granular clustering, we can appreciate that the optimization phase (and fortunately, the data processing phase as well) of the algorithm still completes within seconds.

7.2 Building a sequential procedure

Once we perform the clustering and choose a batch size, the total quantity of each cluster will determine how it will be treated:

- Clusters with a total quantity smaller than the batch size are left out the assignment
- Clusters with a total quantity bigger than the batch size have to be split in portions with total quantity equal to the batch size.

  This will generate some residuals composed by orders entirely or partially discarded from that assignment.

In order to use most of the orders we designed a sequential procedure that iterates through decreasing batch sizes, incorporating residual orders from the previous assignments and grouping them into smaller clusters.

Batch Size	Number of Unassigned Orders	Residual Demand Quantity	% of Inevaded Demand
100000	53133	36402284	51.77
75000	50657	31302284	44.52
50000	47222	24252284	34.49
25000	41300	18977284	26.99
10000	29224	13142284	18.69
7500	26556	16392284	23.31
5000	23731	20862284	29.67
2500	17384	30317284	43.12

Table 3: Inevaded Demand for different Batch sizes

In 3 we can appreciate the different performance both in terms of number of orders and of total demand.
The best performance obtained is 18.69% of not assigned demand at batch size 10000.
We can now confront it with our pipeline that goes from an initial batch size of 10^5 to 50.

Residual Demand Quantity	% of Inevaded Demand	Number of Unassigned Orders
252434	0.3591	4939

Table 4: Pipeline Performance

4 shows that the method is effective, leaving only 0.36% of the total demand unassigned.
Therefore we decided to go on with this pipeline method and treat complexity constraints with it.

7.3 Including complexity constraints

In our first model we ignored the technological constraint that prevents plants from producing unlimited product types.

7.4 Optimization Model with Complexity Constraints

$$\min_{z,b,v,m} \sum_{i,y} \mathbf{CostProd}\, z_i^y + \sum_{i,y} \mathbf{CostSurplus}\, v_i^y + \sum_{j,y'} \mathbf{CostDelay}(y(j), y')\, b_j^{y'} + \sum_{j} \mathbf{CostInevasion}\, m_j$$

$$\text{s.t.} \quad z_i^y \le \mathbf{C}_i^y + \mathbf{EX}_i^y \qquad \forall\, i, y$$

$$v_i^y \ge z_i^y - \mathbf{C}_i^y \qquad \forall\, i, y$$

$$z_i^y = \sum_k z_i^{ky} \qquad \forall\, i, y$$

$$b_j^{y'} \le \sum_i z_i^{k(j)y'} \qquad \forall\, j, y'$$

$$\mathbf{D}_j \le m_j + \sum_{y(j)<y'} b_j^{y'} + \sum_i z_i^{k(j)y(j)} \qquad \forall\, j$$

$$z_i^{ky} \le \mathbf{M}\, u_i^{ky} \qquad \forall\, i, k, y$$

$$\sum_k u_i^{ky} \le \mathbf{Q}_i^y \qquad \forall\, i, y$$

$$u_i^{ky} \in \{0, 1\} \qquad \forall\, i, k, y$$

$$z_i^{ky} \ge 0 \qquad \forall\, i, k, y$$

$$b_j^{ky'} \ge 0 \qquad \forall\, j, k, y'$$

$$m_j \ge 0 \qquad \forall\, j$$

7.4.1 Variable Introduced in the new Problem

- u_i^{ky} :

  This binary variable indicates whether we associate the product type k to plant i at month y.

7.4.2 Hyper-parameters in the new Problem

- $\mathbf{Q}_i^y$:

  This hyper-parameter indicates the maximum different types that can be associated,therefore produced, by plant i at time y.
  Each plant has a different $\mathbf{Q}$ in every month y

- $\mathbf{M}$:

  $\mathbf{M}$ is a large enough value needed for linking the production of a given product type k to the just introduced variable u.

7.4.3 Constraints for the new Problem

- $\sum_k u_i^{ky} \leq \mathbf{Q}_i^y$:

  This constraint ensures that a plant does not produce more product types than its maximum allowable limit.

- $z_i^{ky} \leq \mathbf{M} u_i^{ky}$:

  With $\mathbf{M}$ set sufficiently large, this constraint specifies that product type k can be produced in plant i at time y only after having associated the product.

The fourth and fifth constraints were modified with respect to the previous optimization problem.

- **Constraint 4**
  The portion of order j we decide to produce at any time y' beyond the expected time $y(j)$ will increase the future requested production for the product type of the order.

- **Constraint 5**
  The quantity specified in order j, denoted as $\mathbf{D}_j$, is divided into three parts:

  - A portion of the order which is not produced m_j
  - A portion of the order evaded with delay $b_j^{y'}$
  - A portion of the order evaded in time at $y(j)$.
    This last portion depends on the overall production of good $k(j)$ across all the plants $\sum_i z_i^{k(y)y(j)}$.

7.5 Tackling complexity constraints using a Loss Function

The method we used to prevent the algorithm from going outside the feasible set is the **penalty method**. This technique consists in deleting the hard constraint related to the technological capabilities of plants and adding a penalty component to the objective function.
The resulting model can be formulated as follows:
As we want to leverage the Hungarian algorithm, we tried to model the cost matrix taking in account both the complexities of the units to assign and the maximum number of types that each plant-month can contain.

7.5.1 Optimization Model with Complexity Loss Function

$$\min_{z,b,v,m} \sum_{i,y} \mathbf{CostProd}\, z_i^y + \sum_{i,y} \mathbf{CostSurplus}\, v_i^y + \sum_{j,y'} \mathbf{CostDelay}(y(j), y')\, b_j^{y'} + \sum_{j} \mathbf{CostInevasion}\, m_j + \sum_{i,y} \mathbf{Loss}(\mathbf{Q}_i^y, \sum_k u_i^{ky})$$

$$\text{s.t.} \quad z_i^y \leq \mathbf{C}_i^y + \mathbf{EX}_i^y \qquad \forall\, i, y$$

$$v_i^y \geq z_i^y - \mathbf{C}_i^y \qquad \forall\, i, y$$

$$z_i^y = \sum_k z_i^{ky} \qquad \forall\, i, y$$

$$b_j^{y'} \leq \sum_i z_i^{k(j)y'} \qquad \forall\, j, y'$$

$$\mathbf{D}_j \leq m_j + \sum_{y(j)<y'} b_j^{y'} + \sum_i z_i^{k(j)y(j)} \qquad \forall\, j$$

$$z_i^{ky} \leq \mathbf{M} u_i^{ky} \qquad \forall\, i, k, y$$

$$u_i^{ky} \in \{0, 1\} \qquad \forall\, i, k, y$$

$$z_i^{ky} \geq 0 \qquad \forall\, i, k, y$$

$$b_j^{ky'} \geq 0 \qquad \forall\, j, k, y'$$

$$m_j \geq 0 \qquad \forall\, j$$

In the last component of our new objective function we inserted the term $\mathbf{Loss}(\mathbf{Q}_i^y, \sum_k u_i^{ky})$, function of both $\mathbf{Q}_i^y$, the maximum types that the plant i can produce in month y, and $\sum_k u_i^{ky}$,the amount of different product types assigned.
We removed the hard constraint, giving the opportunity to each plant to produce more types than what are its theoretical limits.
The **Loss** element to the objective function related to the loss pushes the algorithm towards a solution that minimizes the amount of plants producing more product types than they theoretically can.
This technique is not final, as the solutions obtained still present plants violating their technological constraint.
Nonetheless,the results derived from this algorithm variant showed a strong improvement, proving this technique as useful.

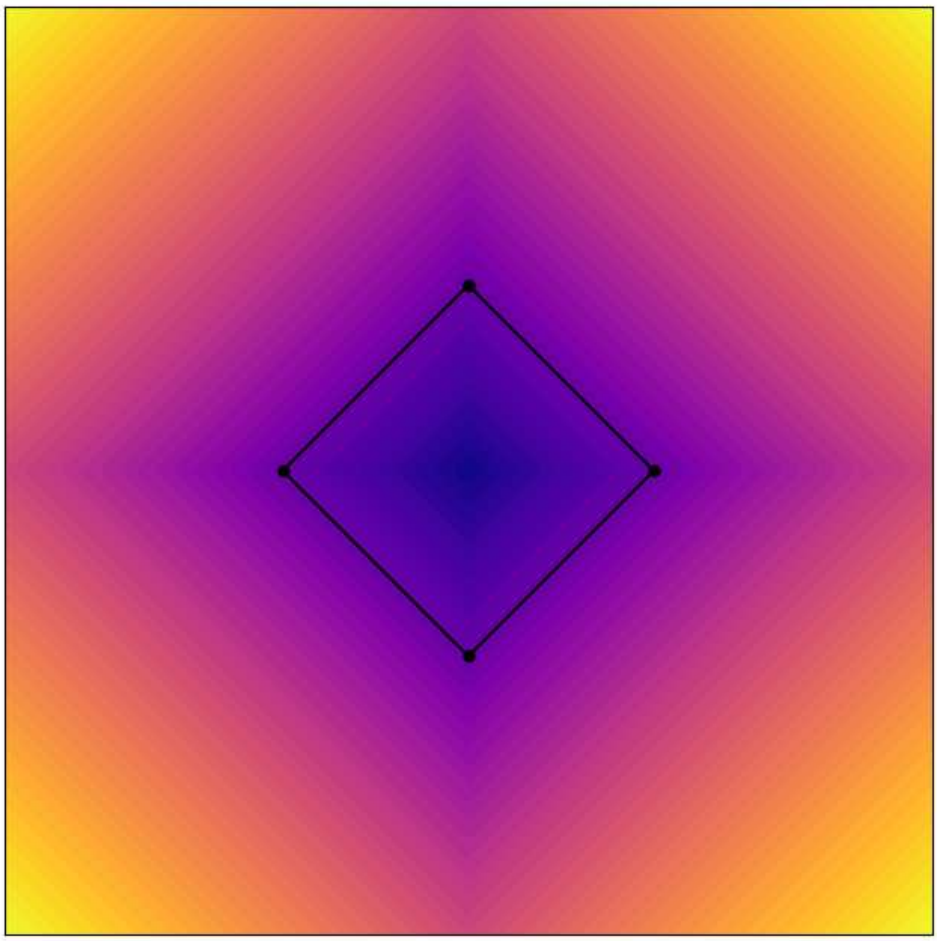

Figure 5: Loss Visualization

Figure 5 illustrates the behavior of the loss function.
It increases progressively as we move toward the exterior boundaries of the feasible set.

7.5.2 New Cost Structure

To adapt the Hungarian algorithm for this new optimization problem, we modified the costs $\mathbf{C}_{ij}$ within the cost matrix.
Our revised cost construction is as follows:

$$\mathbf{C}_{ij} = \mathbf{Cp}(j) + \mathbf{Ct}(i, j) + \mathbf{Cc}(i, j)$$

While the first two terms are as defined in 6.1, we have added a complexity penalty, $\mathbf{Cc}(i, j)$, which depends on:

- The maximum number of new product types that can be assigned to the plant and month associated with column j.
- The different product types included in the element associated with row i.

Along the different iterations of the pipelines, we inspect what types were inserted in the previous rounds and formulate a complexity penalty based on how many new types we are introducing and how many other types we can introduce.
The results are very promising:

Penalty	% Inevaded	Exceeded Complexity	Best Overcomplexity	Worst Overcomplexity
Yes	0.2903	17625	1	393
No	0.3591	28133	2	483

Table 5: Performance with and without complexity penalties

- The total amount of product types that make plants go in overcomplexity decreased from 28133 to 17625
- Between the plants that go in overcomplexity the worst case exceeds by 393 types instead of 483.

8 Initial Exploratory Analysis

A first analysis is of help in order to understand the data we are treating.
The first thing we analyzed was the distribution of different product types along the orders.

- **66024** different orders
- **3782** different product types
- **70302284** total amount of demand

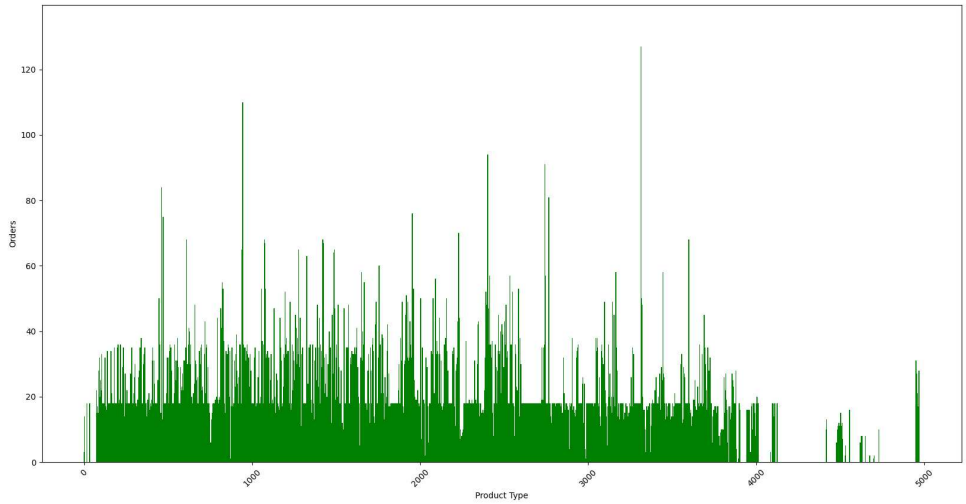

Figure 6: Orders distribution

In Figure 6 we can observe the plot illustrating the frequency of each product type across various orders.

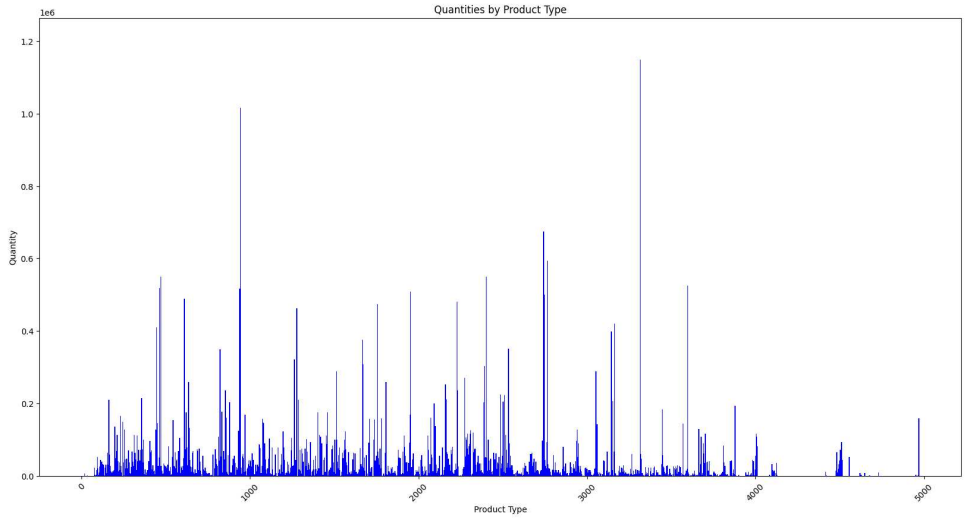

Figure 7: Quantity distribution

Plot 9 shows the amount of orders belonging to different product types.
Among all the types we observe that:

- 7% types have an amount less than 10^2
- 17.8% of types have a total amount between 10^2 and 10^3
- 40% of types have a total amount between 10^3 and 10^4
- 31.3% of types have a total amount between 10^4 and 10^5
- 3% of types have a total amount between 10^5 and 10^6
- 2 types have an amount bigger than 10^6

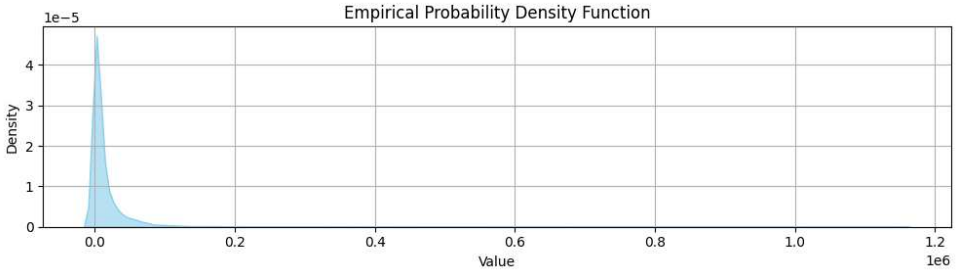

Figure 8: Empirical Quantity Probability Density Function

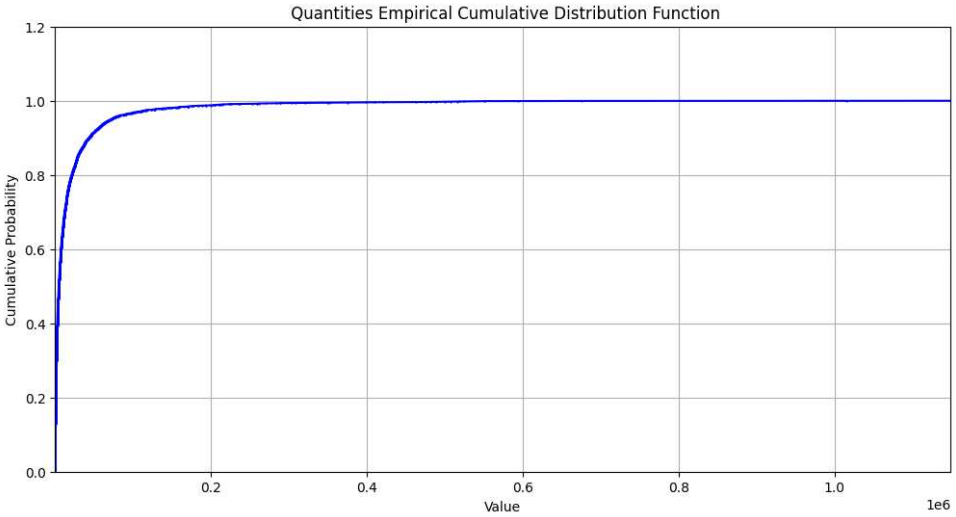

Figure 9: Empirical Quantity Cumulative Density Function

9 Results

We now report the results obtained from different variants of the algorithm.

9.1 Iterative Algorithm without Complexity Penalties

In the first version of our procedure, we did not take in account the complexity constraint, and we only tried to minimize

- Amount of not evaded orders
- Amount of not orders evaded with delay

During the creation of the cost matrix, as each row identifies a specific cluster of orders with its own specific expected production month and compatibilities, while iterating along different capacity units, we had to decide how to formulate the different costs capable of pushing the assignment in the right direction.
We decided to set the following hierarchies between different costs:

- Using extra capacity slots costs more than default capacity
- To account for delays, we introduced a delay penalty function that increases as delays grow. Each cluster unit has a designated expected production month, and the penalty escalates progressively the further the actual production deviates from this expected timeline
- Similarly, we aimed to minimize the number of orders produced in advance to develop a final solution that requires minimal warehouse space
- We structured the anticipation and delay penalties so that producing an order just 1 month late incurs a greater penalty than producing it up to 18 months in advance.

We tried different combinations to test the algorithm limits over this instance.
After extensive trials focused on penalty tuning, we identified two key configurations.

9.1.1 Linear penalties

The first idea we had was to implement a linear penalty for delays and anticipations.
Again, as we care more about the delays, we made sure that the penalties behaved accordingly.

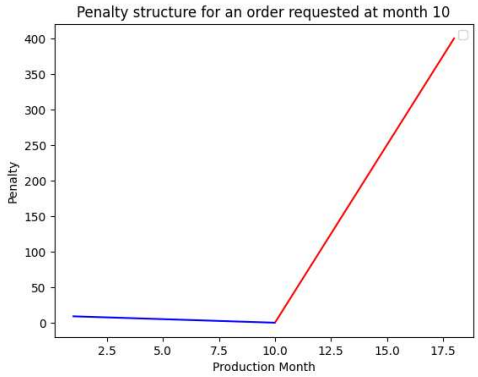

Figure 10: Empirical Quantity Probability Density Function

Figure 10 shows the piecewise linear function we used.
The red part is related to the delay penalty that, for the reasons we just stated, grows faster than the anticipation.
The final function is:

$$f_m(x) = K_a(m-x)1_{\{x \leq m\}} + K_d(x-m)1_{\{x \geq m\}}$$

9.1.2 Polynomial Penalties

To try and reduce the amount of demand evaded in delay and the average delay, we decided to test where the algorithm would bring us after setting a more severe penalty associated with the delay.
We opted for a quadratic delay penalty.

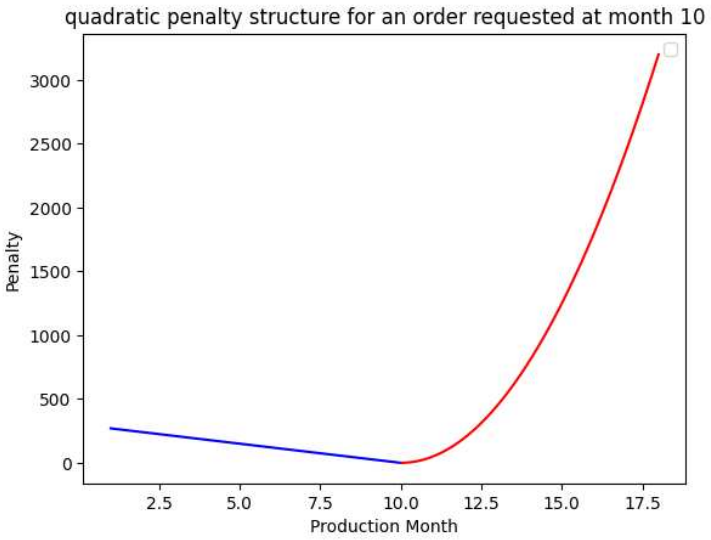

Figure 11: Empirical Quantity Probability Density Function

The penalty function pictured in 11 can be formulated as follows:

$$f_m(x) = K_a(m-x)1_{\{x \leq m\}} + K_d(x-m)^2 1_{\{x \geq m\}}$$

9.1.3 Results and Parameters

The hyperparameters that emerged as the best performers during the tuning phase are:

Default Capacity Cost	1
Extra Capacity Cost	5
K_a	1
K_d	50

Table 6: Penalty function hyper-parameters

Using these values we obtained the following results:

Penalty	% Inevaded Demand	% Demand in Delay	% Demand in Anticipation
Linear	0.337	1.2	0.22
Quadratic	0.337	1.6	0.22

Table 7: Performance with and without complexity penalties

Penalty	Avg.Delay	Avg.Anticipation
Linear	1.5	1.27
Quadratic	1.15	1.27

Table 8: Performance with and without complexity penalties

The results are promising, with only about 0.33% of the total demand remaining unassigned, and the percentage of demand assigned with delays is around 1-2%.
We observe that a more aggressive delay penalty steers the algorithm toward a smaller average delay; however, this comes at the cost of a larger portion of demand being assigned after the expected month. The term in the objective function related to the delay costs

9.2 Iterative Algorithm with Penalties

After the first tries we started taking into account the complexity constraints.
We inserted a complexity penalty based on the ratio between the different types we are trying to assign and the remaining types that can be inserted in the plant.
After applying different transformations to this core value we monitored how the obtained solutions reacted.
From now on we will call Core(i, j) the ratio $\frac{\text{new set of types in cluster } \mathbf{i} \text{ I am inserting}}{\text{maximum types the plant } \mathbf{j} \text{ can still host}}$.

9.2.1 Linear Complexity Penalty

The first complexity penalties we tested were linear functions

$$\mathbf{Penalty}_{(i,j)} = K \cdot \mathbf{Core}_{(i,j)}$$

With K as a variable hyper-parameter.

9.2.2 Soft-Max Based Penalty

After the two baseline functions we experimented using soft-max.

$$\mathbf{Penalty}_{(i,j)} = \frac{M}{1 + e^{-\lambda Core_{(i,j)}}}$$

where :

- M is a hyper-parameter representing the maximum possible penalty value.
- λ controls the steepness of the sigmoid function.

We set a maximum penalty value to use when we meet plants having already exceeded their complexity constraint.
Using the sigmoid function gives us the opportunity to apply a penalty that approaches the maximum penalty value M and using various steepness values λ it is possible to experiment more or less aggressive complexity penalties.

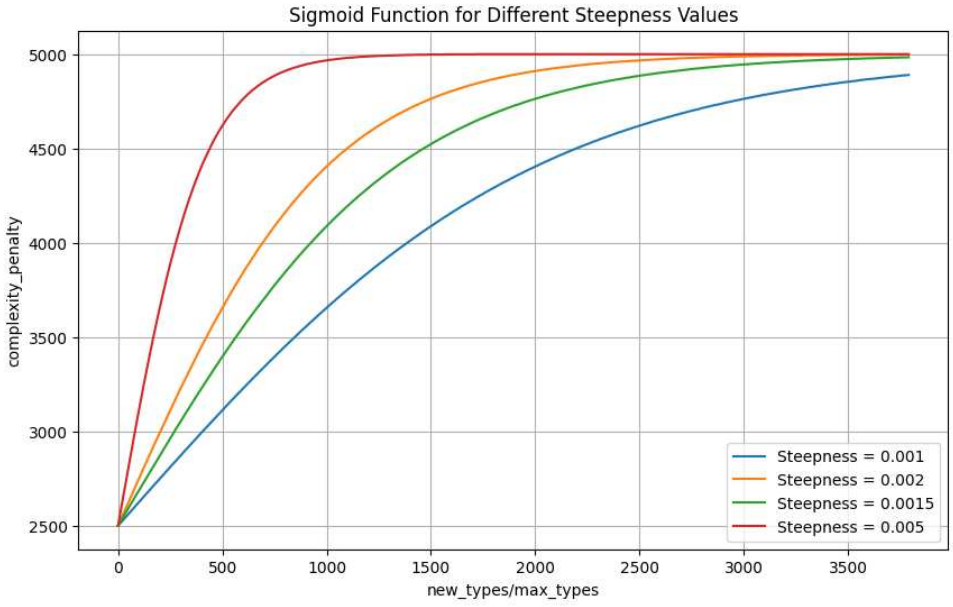

Figure 12: Complexity Penalty function with Different Steepness Values

We can appreciate how, once set a maximum penalty value, it was possible to play with the steepness hyper-parameter and see how different levels of aggression affected the final solution.

Complexity Exceedance	19759
Smallest Exceednace	3
Biggest Exceedance	315

Table 9: Results Obtained without Complexity Penalties

The results just showed refer to the algorithm version without any complexity constraints.

Linear Coefficient K	1	10
% Inevaded Demand	0.33	0.33
% Demand in Delay	1.45	1.67
% Demand in Anticipation	5.78	6.68
Avg. Delay	1.49	1.48
Avg. Anticipation	2.39	2.48
N.of Types in Exceedance	14732	14118
Smallest Exceedance	1	1
Biggest Exceedance	244	269

Table 10: Performance with Linear and Quadratic Complexity Penalty

In 10, the results of the algorithm incorporating linear complexity constraints are presented.
We can observe that, to remain within the feasible complexity set, the proportion of demand evaded with delay increased significantly from approximately 0.22% to 1.45-67%.
As for the distribution of different types, we pleasantly notice that the amount of types assigned in exceedance reduced from about 19000 to 14000. The sigmoid approach proved to be the most effective.

Steepness	0.001	0.002	0.005	0.01	0.05	0.5
% Inevaded Demand	0.337	0.337	0.337	0.34	0.35	0.3
% Demand in Delay	1.42	1.49	1.53	1.64	1.81	4.63
% Demand in Anticipation	6.29	8.65	13.13	16.2	24.67	30.55
Avg. Delay	1.52	1.5	1.49	1.46	1.35	1.38
Avg. Anticipation	2.52	1.99	1.66	1.71	2.32	3.33
N.of Types in Exceedance	14507	13378	11380	10141	7943	6842
Smallest Exceedance	1	1	1	1	1	1
Biggest Exceedance	273	245	246	210	213	195

Table 11: Performance of Sigmoid Complexity Penalty

Not only did it improve compliance with the complexity constraints, but it also unexpectedly enabled the algorithm to keep delays under control.
Increasing the steepness hyper-parameter we managed to reduce the total exceedance to 6842, but this came at the price of an higher percentage of demand evaded in delay.
We went on and explored some more aggressive steepness values.

Steepness	1	2	3	5
% Inevaded Demand	0.3	0.29	0.3	0.29
% Demand in Delay	6.5	9.01	9.46	9.85
% Demand in Anticipation	30	29.06	29.6	29.96
Avg. Delay	1.37	1.47	1.48	1.46
Avg. Anticipation	3.57	3.68	3.7	3.9
N.of Types in Exceedance	6590	6759	6956	7220
Smallest Exceedance	1	1	1	1
Biggest Exceedance	195	192	196	206

Table 12: Performance of Sigmoid Complexity Penalty with High Aggressiveness

Using aggressive steepness values was revealed not to be the ultimate solution as beyond value 1 the solution started showing an increase in complexity exceedance.

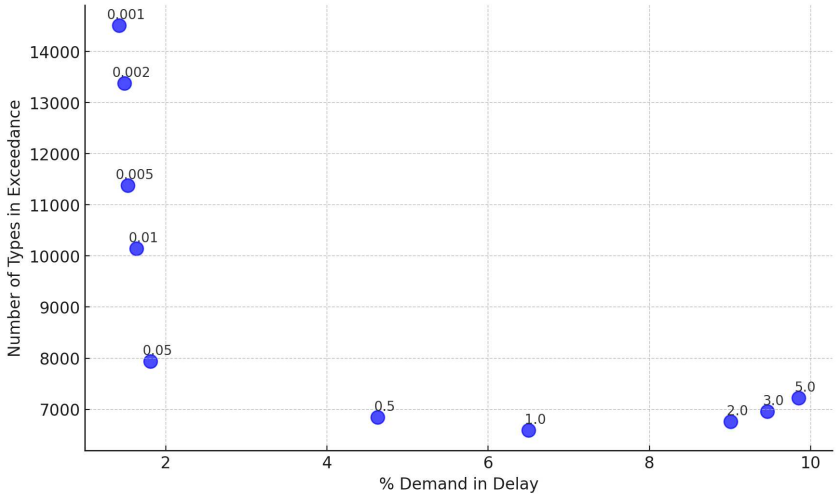

Figure 13: Steepness effect on Delay and Types in Exceedance

In order to minimize the instances where complexity exceeds acceptable thresholds, the algorithm is forced to increase the proportion of orders assigned to delay slots. This trade-off allows the system to better manage capacity constraints and complexity requirements, even though it results in a higher percentage of delayed orders.

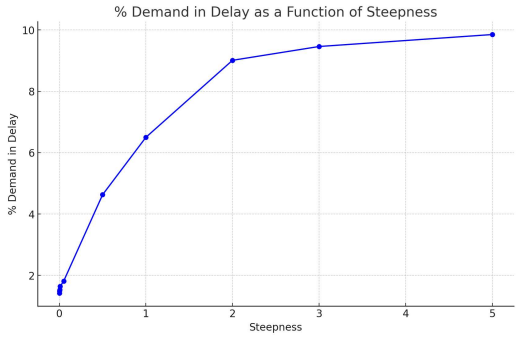

Figure 14: Demand Evaded in Delay

9.3 Reallocating Orders to Prevent Excessive Splitting

To transform the problem into an assignment problem, we first ensured that each element in the rows of the cost matrix was of the same size as the capacity elements in the columns.
For each iteration, we set a specific batch size (e.g., 10,000, 5,000, etc.). After performing clustering, the rows contained a total demand amount that often exceeded the batch size due to the varying sizes of individual orders. To address this, we split the last order in each row, including only a portion of it to meet the batch size limit.
As a result, some orders frequently appear across multiple iterations, reflecting their partial allocation over different batches and often different plants.
We want to obtain a production setting where an order is assigned to a plant in its entirety.
In order to do so we implemented an improvement method that at the end of the assignment takes all the orders that were split and assigns the left out portion to the plant where the included part was assigned.

Steepness	0.001	0.002	0.005	0.01	0.05	0.5
% Inevaded Demand	0.30	0.30	0.30	0.30	0.26	0.27
% Demand in Delay	1.62	1.66	1.63	1.70	1.99	4.6
% Demand in Anticipation	6.33	8.87	13.19	16.3	24.5	30.22
Avg. Delay	1.46	1.46	1.46	1.40	1.35	1.37
Avg. Anticipation	2.52	1.84	1.65	1.79	2.28	3.36
N. of Types in Exceedance	13838	12310	10336	8926	7429	6407
Smallest Exceedance	1	1	1	1	1	1
Biggest Exceedance	266	222	201	184	207	182

Table 13: Performance of Sigmoid Complexity Penalty

The trend did not change, we can still notice that pushing the algorithm towards smaller complexity exceedance it has to increment the amount of orders in delay.
As before, we study the performances obtained using higher penalties.

Steepness	1	2	3	5
% Inevaded Demand	0.29	0.28	0.29	0.28
% Demand in Delay	6.6	8.93	9.54	9.68
% Demand in Anticipation	30	29.5	29.66	30.12
Avg. Delay	1.37	1.47	1.49	1.47
Avg. Anticipation	3.67	3.84	3.84	3.86
N.of Types in Exceedance	6193	6493	6818	6703
Smallest Exceedance	1	1	1	1
Biggest Exceedance	180	195	199	226

Table 14: Performance of Sigmoid Complexity Penalty with High Aggressiveness

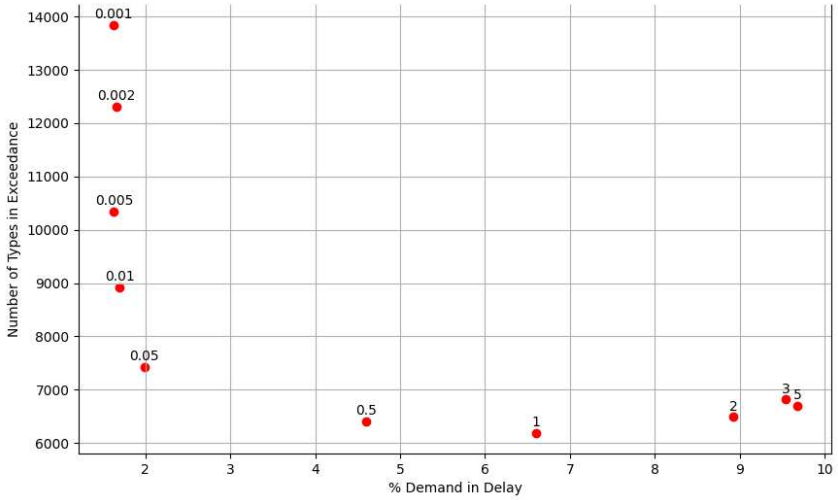

Figure 15: Steepness effect on Delay and Types in Exceedance with reassemble method

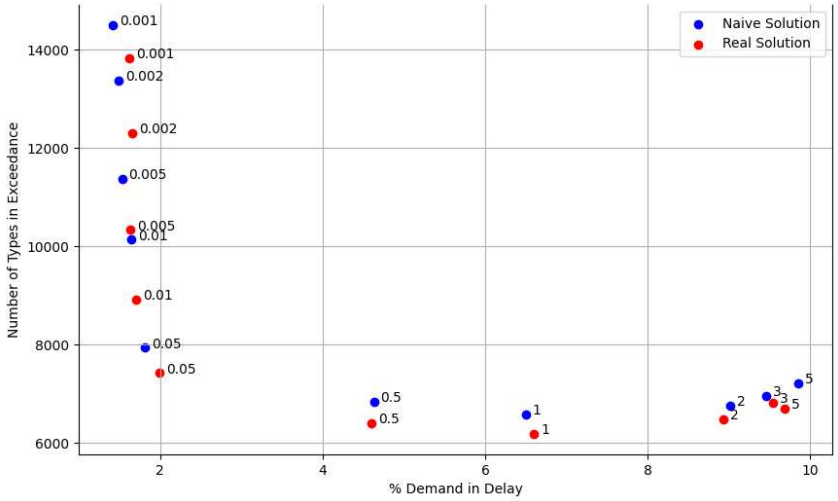

Figure 16: Steepness effect on Delay and Types in Exceedance with reassemble method

In 15 we can assert that the solution obtained with this new correction method closely resembles the previous one in terms of performance.
The plot in 16 is encouraging, as it suggests that achieving a simpler solution, where each order is fully assigned to a single plant, not only maintains performance but can even lead to improved results.

10 Conclusions

The goal of our work was to create a baseline optimization algorithm capable of generating solutions that effectively satisfy the key constraints requested by the client.
In the first phase of our work we understood that the most problematic part of the main problem was the production optimization so we focused our research on it.
After analyzing the problem, we decided to model and approach it as a classical assignment problem, incorporating all the main constraints into the model. This allowed us to reduce the optimization problem to a form suitable for leveraging the polynomial complexity of the Hungarian algorithm.
Through this approach, we obtained a set of solutions that align with the client's requirements with an high level of computational efficiency.
Our algorithm consistently generates solutions within minutes, despite the large scale of the instance we worked on.
The solutions obtained from our approach exhibit a low percentage of unfulfilled orders, indicating that the algorithm can reliably meet demand under typical operating conditions.
Additionally, the algorithm effectively minimizes the number of plants that exceed the maximum product type constraint, a critical factor in maintaining efficient operations and ensuring regulatory compliance in production environments. This balance between satisfying demand and respecting plant constraints showcases the adaptability and effectiveness of our solution in real-world applications.
This method provides a foundational approach that paves the way for more advanced methods to handle complex constraints.
Starting from a consistent solution generated by our baseline tool, further optimized solutions can be obtained through a combination of classical solvers and metaheuristic techniques.

11 Algorithm Explained in Detail

```
def process_batch_data5(plant_data, demand_data, brow, round_number,
    complexity_penalty,reallocate_to_make_integer):

    current_batch = brow[round_number]

    plant_data = batch_plant(plant_data, current_batch)
    cap, kap, total_capacities, checkpoint = your_function(plant_data,
        current_batch)

    demand_data, dfb, quantity_taken, quantity_not_taken =
        generate_clustered_demand(demand_data, current_batch)
    if complexity_penalty:
        start_time = time.time()
        compatibility_matrix = generate_matrix4(plant_data, dfb, cap,
            checkpoint, total_capacities, brow, round_number)
        end_time = time.time()
    else:
        start_time = time.time()
        compatibility_matrix = generate_matrix2(dfb,current_batch, cap,
            checkpoint, total_capacities,True)
        end_time = time.time()


    time_of_creation=end_time - start_time##
    demand_unit=compatibility_matrix.shape[0]##
    capacity_unit=compatibility_matrix.shape[1] ##


    compatibility_matrix = compatibility_matrix.astype(int)
    compatibility_matrix, unassignment_cost = analyze_matrix(
        compatibility_matrix, current_batch, 1_000_000)

    T, row_ind, col_ind, costs, total_cost = hungarian(compatibility_matrix)

    assignment_map, assignment_map_orders = assignment_maps(dfb, col_ind, costs
        , unassignment_cost, checkpoint, kap)

    demand_data = process_demand_df(demand_data, assignment_map_orders,
        current_batch)
    updated_plant_data = update_plant_data4(plant_data, assignment_map,
        assignment_map_orders, demand_data, brow, round_number)

    if reallocate_to_make_integer:
        print(f"Reallocation at {brow[round_number]} initialized")
        demand_data,updated_plant_data,integration_map=reallocate_orders(
            demand_data, updated_plant_data, brow, round_number)
    else:
        print("We keep the orders split")
        integration_map=None

updated_plant_data,demand_data,current_batch,demand_unit,capacity_unit,
    time_of_creation,T,memory_usage,integration_map
```

Listing 1: Main Processing Algorithm

The inputs of the main function are the following:

- **plant_data, demand_data** : Both the datasets related to orders and plants
- **brow, round_number** : A list containing the batch sizes to be used sequentially in the algorithm, along with the current round number within the sequence.
- **complexity_penalty**: A boolean to specify whether we want to add the complexity penalties discussed in 9.2
- **reallocate_to_make_integer** A boolean to specify whether we want to perform the reallocation discussed in 9.3 to minimize the split orders along the iterations.

11.1 Processing the Plant Dataframe

The first thing we do is divide the plants into units with equal capacities, which can then be placed as columns in the cost matrix.

```
batch_plant(plant_df, 10000)
```

Listing 2: Plant Batching Algorithm

plant_id	month	capacity	extra_capacity	capacity columns	extra capacity columns
0	1	20479	2048	2	0
1	1	292185	29219	29	2
2	1	430222	43022	43	4
3	1	1261253	126125	126	12
4	1	37219	3722	3	0
...	...	...	...	...	...
14	18	336192	33619	33	3
15	18	693700	69370	69	6
16	18	96800	9680	9	0
17	18	477972	47797	47	4
18	18	9658	966	0	0

Figure 17: Splitting of the Plant Capacities

Each plant in each month is represented by a set of columns inside the cost matrix.
For example, the first row of Figure 17 illustrates that the plant 0 at the first month has a maximum capacity of 20479 and an extra capacity slot of 2048.
Given that we chose to use column elements with a capacity of 10,000, this plant will be represented by two columns in the cost matrix, each corresponding to a portion of the plant's default capacity.
As for the second row the result is different because the extra capacity slot of 29219 will be represented by two columns in the cost matrix.

11.2 Processing the Orders Dataframe

```
demand_data, dfb, quantity_taken, quantity_not_taken =
    generate_clustered_demand(demand_data, current_batch)
```

Listing 3: Clustering the Orders

This stage of data processing primarily aims to create a dataframe in which each row will correspond to a row element in the cost matrix.

11.2.1 Grouping the Orders

As stated in 5 we grouped all the orders by the production compatibility.

prod_month	quantity	in time	in delay
12	30013	[[2, 1], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6...	[[2, 13], [2, 14], [3, 13], [3, 14], [15, 13],...
15	30105	[[4, 1], [4, 2], [4, 3], [4, 4], [4, 5], [4, 6...	[[4, 16], [4, 17]]
3	30338	[[6, 1], [6, 2], [6, 3], [11, 1], [11, 2], [11...	[[6, 4], [6, 5], [11, 4], [11, 5]]
14	30442	[[6, 1], [6, 2], [6, 3], [6, 4], [6, 5], [6, 6...	[[6, 15], [6, 16], [11, 15], [11, 16], [15, 15...
16	30660	[[3, 1], [3, 2], [3, 3], [3, 4], [3, 5], [3, 6...	[[3, 17], [3, 18], [15, 17], [15, 18]]
...	...	...	...

Figure 18: Grouping the Orders

The DataFrame in Figure 18 displays all the clusters obtained. Each entry represents a cluster associated with specific **in time** and **in delay** values.

11.2.2 Divide the rows in elements of the same quantity

The DataFrame in Figure 18 contains clusters of varying sizes: some are smaller than the batch size, while others are larger. For the smaller clusters, we discarded them. For clusters exceeding the batch size, we applied the following steps:

- We grouped all orders within each cluster into subsets, each with a total quantity of at least the desired batch size.
- This process will likely produce subclusters with quantities slightly exceeding the batch size due to the addition of the last order, as well as a final subcluster with a quantity smaller than the batch size.
  In the first case, we split the last added order, retaining only the exact amount needed to make the subcluster's quantity equal to the batch size
  For the remaining subclusters smaller than the batch size, we discarded them entirely.

At the end of this operation we obtain a dataframe on length m that will become the height of the cost matrix.

11.2.3 Update the Order Dataframe

We use the order dataframe to store the information abut the order history.
Based on the operations described above an order can be:

- Excluded from the clusters, therefore from the assignment.
- Partially included, if it happened to be the last order inside the subclusters described in 11.2.2.
- Entirely included in the cluster.

id	product_id	product_type_id	prod_month	quantity	quantity included	quantity excluded
55013	4016	1171	16	13	7	6

Figure 19: Example of a Split Order

Figure 19 shows an example of split order.
Keeping track of the amount included and excluded along iterations helped us to:

- Check eventual updating errors.
- Update the order dataframe at the end of each iteration.

11.3 Creating the Cost Matrix

At this point we have all the row and column elements needed to create the cost matrix.

```
def generate_matrix(plant_data,input_df,capacity_param, check_param,
    total_capacity_size,brow,r):
    min_quantity=brow[r]
    compatibility_matrix = np.zeros((len(input_df), total_capacity_size), dtype
        =int)
    for row_index, (_, row) in enumerate(input_df.iterrows()):

        if row["quantity"] >= min_quantity:

            arr = compatibility_array_with_pen(
            plant_data,
            row["n.types"],
            row["group1"],
            row["prod_month"],
            row["in time"],
            row["backorder"],
            capacity_param,
            check_param,
            total_capacity_size,
            brow,
            r,
            True)
        compatibility_matrix[row_index, :] = arr
    return compatibility_matrix
```

Listing 4: Generating the Cost Matrix

The function takes in input both the processed dataframes we just analyzed.
After initializing an empty matrix, for each row of the clustered orders dataframe, a cost array is created iterating along the columns, taking into considerations the type of capacity, the delays and the complexity penalties discussed in 6.1 and 7.5.2.

11.4 Costs and Penalties assignment

The following code snippet demonstrates the calculation of penalties and the process of setting varying costs across each row of the cost matrix:

```
def compatibility_array_with_pen(plant_data, n_types, row_types, precise_month,
    in_time_list, in_delay_list, cap_mat, checks, total_blocks, brow, r):

    maxi = 5000
    s = 5
    arr = np.zeros((1, total_blocks))

    for x in in_time_list:
        index = plant_index2(x)

        max_types = plant_data.at[index, "max_product_types"]
        if max_types <= 0:
            complexity_penalty = maxi
        else:
            if r == 0:
                complexity_penalty = int(maxi / (1 + np.exp(-s * (n_types /
                    max_types))))
            else:
                types_scan = plant_data.at[index, f"scan {brow[r - 1]}"]
                types_in_plant = [sublist[0] for sublist in types_scan]
                types_in_demand = [sublist[0] for sublist in row_types]
                new_types = list(set(types_in_demand) - set(types_in_plant))
                core = len(new_types)
                complexity_penalty = int(maxi / (1 + np.exp(-s * (core /
                    max_types))))

        anticipation = precise_month - x[1]
        coord = coordinates(x, cap_mat, checks)

        arr[0, coord[0]:coord[1]] = 1 + anticipation + complexity_penalty
        arr[0, coord[1]:coord[2]] = 5 + anticipation + complexity_penalty

    for y in in_delay_list:
        index = plant_index2(y)
        max_types = plant_data.at[index, "max_product_types"]

        if max_types <= 0:
            complexity_penalty = maxi
        else:
            if r == 0:
                complexity_penalty = int(maxi / (1 + np.exp(-s * (n_types /
                    max_types))))
            else:
                types_scan = plant_data.at[index, f"scan {brow[r - 1]}"]
                types_in_plant = [sublist[0] for sublist in types_scan]
                types_in_demand = [sublist[0] for sublist in row_types]
                new_types = list(set(types_in_demand) - set(types_in_plant))
                core = len(new_types)
                complexity_penalty = int(maxi / (1 + np.exp(-s * (core /
                    max_types))))

        delay = y[1] - precise_month
        koord = coordinates(y, cap_mat, checks)

        arr[0, koord[0]:koord[1]] = 1 + 50 * delay + complexity_penalty
        arr[0, koord[1]:koord[2]] = 5 + 50 * delay + complexity_penalty
    return arr
```

Listing 5: Setting Costs along the Cost Matrix Columns

This code demonstrates the process of creating individual rows of the cost matrix.
Here, we are working with a single row element in the cost matrix, with a unique set of plants and months where production can occur either on time or with delay.

- We initialize the row creating an array filled with zeros. **(5)**
- We iterate through all possible options within this set. **(7,30)**
- For each option, we check which product types have already been assigned to it. **(17-22,40-45)**
- In the first iteration of the algorithm, we directly assess the maximum number of different types that each plant can produce. **(14,37)**
- We calculate the complexity penalty to associate. **(15,22,38,45)**
- Depending on the element under consideration, we compute either the delay or anticipation **(24,47)**
- We populate the array at the indices corresponding to the element under examination with the final computed cost.
  A noteworthy detail is that, when assigning costs to the array indices, we encounter two distinct sets of indices:**(27-28,50-51)**
  - The first set corresponds to column indices representing the default capacity of the element being examined.
  - The second set corresponds to indices representing the extra capacity

11.5 Processing the Results of the Assignment

```
1     T, row_ind, col_ind, costs, total_cost = hungarian(compatibility_matrix)
2
3     assignment_map, assignment_map_orders = assignment_maps(dfb, col_ind, costs
          , unassignment_cost, checkpoint, kap)
4
5     demand_data = process_demand_df(demand_data, assignment_map_orders,
          current_batch)
6     updated_plant_data = update_plant_data4(plant_data, assignment_map,
          assignment_map_orders, demand_data, brow, round_number)
```

Listing 6: Update the dataframes

After obtaining the raw results from the Hungarian algorithm in row 1—expressed in terms of rows and columns—we interpret these results to determine which row element was assigned to which plant, in which month, and with which type of capacity. We then review the orders within the assigned row elements and update both the plant and the order dataframe accordingly.

11.6 Reassemble Split Orders

```
    if reallocate_to_make_integer:
        print(f"Reallocation at {brow[round_number]} initialized")
        demand_data,updated_plant_data,integration_map=reallocate_orders(
            demand_data, updated_plant_data, brow, round_number)
    else:
        print("We keep the orders split")
        integration_map=None
```

Listing 7: Update the dataframes

This part of the algorithm performs the reassembly of split orders, as described in 9.3.
At the end of each round in the sequential procedure, the algorithm checks, for each split order, whether the plant to which it was assigned has remaining capacity.
If so, it adds the residual portion of the order to that plant.

References

[1] Munkres, J. (1957). Algorithms for the Assignment and Transportation Problems. Journal of the Society for Industrial and Applied Mathematics, 5(1), 32–38. doi:10.1137/0105003

[2] David F. Crouse, "On implementing 2D rectangular assignment algorithms" *IEEE Transactions on Aerospace and Electronic Systems* , vol. 52, no. 4, pp. 1679 - 1696, Month,Year: August 2016.

[3] Burkard, Rainer E., Dell'Amico, Mauro, and Martello, Silvano. *Assignment Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2009.

[4] https://docs.scipy.org/doc/scipy-1.11.4/reference/generated/scipy.optimize.linear_sum_assignment.htmlid2