Design and Optimization Project

Optimizing Supply Chain Order Allocation with Mixed Integer Programming [1]

Paul Grassi, Leander Merbecks, Gaëtan Petri, Valentin Schmeller

Introduction

The purpose of this work is to solve a complex mathematical optimization problem in the supply chain domain. The problem aims to optimize the distribution of orders to customers in a production and shipping network whereby every customer demand shall be satisfied while minimizing total costs. To achieve this, we need to model a problem that integrates the operational, physical and logistical constraints of this situation. This report describes our approach to solving this problem. First, we present the problem, then we propose a mathematical MIP (Mix Integer Programming) model, and finally we present our results. The model is implemented in the Gurobi environment of Python.

1. Definition

As mentioned above, the problem addressed in this project concerns the logistics optimization of a production and shipping network. We have an inventory of customer orders to be manufactured in warehouses and then shipped to the port of destination. It's a problem related to the famous "coupling" problems, where a production plant and a forwarding site have to be linked to an order. So, there's a double optimization problem: minimizing both, production costs and storage costs. For this, we have a large data set in an Excel file. Seven tables are at our disposal and are described below:

- Order list: this table contains the customer's order book. For each order, we have an order ID (Order ID), a product ID (Product ID), the destination port (Destination Port), the customer (Customer), the quantity in units (Unit quantity), and the total weight of the shipment (Weight).
- FreightRates: this table contains the shipping rates for orders, taking into account the origin destination (orig_port_cd) and the destination port (dest_port_cd), as well as the minimum cost and the weight-based (rate).
- WhCosts: this table provides storage costs per unit (Cost/unit) for each plant (WH). This table is directly usable for calculating production costs within our objective function.
- WhCapacities: this table determines the daily production capacities of orders (Product ID) per plant (Plant ID) per day. This data is straightforward, with no adjustments needed.
- ProductPerPlant: this table specifies the products (Product ID) that each plant (Plant Code) can produce, imposing compatibility constraints without requiring modifications.
- VmiCustomers: this table specifies which plants (Plant Code) are authorized exclusively to process orders from certain customers (Customers), adding a constraint to this assignment.

- PlantPorts: this table describes the possible connections between plants (Plant Code) and shipping ports (Port), a key logistical consideration in routing the orders.

The goal of this problem is twofold: to meet a maximum number of customer orders within the limits of possible physical and temporal capacities, while minimizing costs.

2. Data preprocessing

The data set provides essential content for solving the problem, although there is a lack of clarity and some points are ambiguous. To clarify these aspects, we have reviewed the data and taken into account certain assumptions:

First we had to omit all data associated with plant 19 because there was no information on the products plant 19 could produce.

The sheet 'OrderList' provides essential details for each order, including the product (what to ship), destination port (where to ship), customer, unit quantity, and shipment weight. Columns such as order origin, service level, ship-ahead day count, ship-late day count, and plant code are omitted as they are irrelevant to our optimization. Only Order ID, Product ID, Destination Port, Customer, Unit quantity, and Weight are used.

The sheet 'FreightRates' gives shipping rates to the fixed destination port 9 based on different origin ports. In the data multiple carriers are listed as shipping options from the different ports. Some carriers have more than one option for shipping to port 9, where the options differ in the weight limits of the shipment. Because the shipping carrier for each order was not set as a decision variable, the available fixed and variable costs for all carriers shipping from one port to the fixed destination port 9 were averaged using the mean value.

Furthermore, for port 1 there was no shipping cost data given in the sheet 'FreightRates'. Thus, we averaged the fixed and variable cost of all other ports and assigned it to this port.

Lastly, the product capabilities in 'ProductsPerPlant' of "plant" CND9 were omitted as it had no matching data in the other sheets. As the product capabilities of this entry matched with plant 16 no production capability was lost completely.

3. Decision variables

To formulate this problem, we defined two decision variables to control the allocation of orders to factories and ports for days.

1. Decision variable for the plant:

x_{ijd} : Binary variable which is equal to 1 if order i is produced in plant j on day d, 0 otherwise.

This variable will be useful for defining production costs for each order, based on the plant assignment.

2. Decision variable for the port:

y_{ikd} : Binary variable which is 1 if command i is produced in port k on day d, 0 otherwise.

This variable will be used to define the shipping costs for each order, depending on the port selected.

4. Objective function

The objective function for this model is the cost of producing all ordered items and shipping all orders. This cost consists of two main components: the cost of production and the cost of shipping.

a. Cost of Production (CoP)

With the plant allocation decision variable $x_{ijd} \epsilon (0,1)^{N\times M\times D}$, where N is the number of orders, M is the number of plants and D is the number of days, we can define the cost of production for all orders. For this, we also need:

- The warehouse cost vector cpu_j, which specifies the unit cost at each plant.
- The number of items per order vector NoI_i, representing the quantity required for each order i.

The cost of producing order i in plant j is calculated as:

$$cpu_j \cdot NoI_i$$

By including the binary decision variable x_{ijd} for this order and plant, we can calculate the overall production cost by summing over all plants and all orders:

$$CoP(x) = \sum_{d=1}^{D} \sum_{i=1}^{N} \sum_{j=1}^{M} cpu_j \cdot NoI_i \cdot x_{ijd}$$

Figure 1 shows, how this mathematical expression is implemented in python:

```
cost_of_production = gb.quicksum(cost_per_unit[index_plant]*ID_quant_per_order[index_order,1]*
                                 plant_alloc[index_order,index_plant,index_day]
                            for index_plant in range(number_of_plants) for index_order in range(number_of_orders)
                                 for index_day in range(number_of_days))
```

Figure 1: Implementation of 'Cost of Production' in Python

b. Cost of Shipping (CoS)

The cost of shipping (CoS) can be computed in a similar manner, except the cost of shipping is now defined by the rate of the port from which the order is shipped, the weight of the order and the fixed cost for shipping. This requires:

- The port shipping rate vector cpw_k, which defines the rate at each port.
- The number of items per order vector WoO_i, representing the weight of each order i.
- The port fixed shipping cost cf_k, which defines the fixed cost for shipping for port k

The cost of shipping an order i via a port k is:

$$(cf_k + cpw_k \cdot WoO_i)$$

By including the binary decision variable y_{ikd} for shipping this order via that port and summing over all orders and all ports, we get:

$$CoS(y) = \sum_{d=1}^{D} \sum_{i=1}^{N} \sum_{k=1}^{K} (cf_k + cpw_k \cdot WoO_i) \cdot y_{ikd}$$

```
cost_of_shipping = gb.quicksum((costs_per_port[index_port,0]+costs_per_port[index_port,1]*
                                weight_per_order[index_order])*port_alloc[index_order,index_port,index_day]
                                 for index_port in range(number_of_ports) for index_order in range(number_of_orders)
                            for index_day in range(number_of_days))
```

Figure 2: Implementation of 'Cost of Shipping' in Python

c. Penalty summand

To account for VMI (Vendor-Managed Inventory) customers, a constraint must be introduced (for further details, see equality constraints 5c). However, due to the limitations of the given dataset, it is sometimes impossible to fully satisfy these constraints when a VMI contract exists between a customer and one or more plants that do not produce the requested product. To allow constraint violations in these cases, a penalty has been introduced. This penalty captures constraint violations of the constraint ‘VMI Customers’ in the objective function and requires the following parameters:

- A sufficiently high multiplier P, which determines the penalty for each violated constraint
- The binary variable vs_{ij}, which determines for each order i and each plant j, weather it is possible to fulfill the VMI constraints, in case there are any VMI constraints (for further explanation, see chapter equality constraints 5c).

The penalties for all orders therefore can be determined as follows:

$$penalty(vs) = P \cdot \sum_{i=1}^{N} \sum_{j=1}^{M} vs_{ij}$$

```
penalty = vmi_penalty_multiplier * gb.quicksum(vmi_slack[index_order, index_plant]
                                                for index_order in range(number_of_orders)
                                                for index_plant in range(number_of_plants))
```

Figure 3: Implementation of penalty summand for unfulfillable orders of the constraint 'VMI Customers' in Python

d. Optimization Function

As stated, the complete objective function is the sum of the cost of production and the cost of shipping:

$$OF(x, y, vs) = CoP(x) + CoS(y) + penalty(vs)$$

This function is always linear because each term in the sums is a linear combination of a single variable and all operations are simple linear additions. This ensures that the model maintains its linearity, making it suitable for a MILP approach.

```
cost_total = cost_of_production + cost_of_shipping + penalty
```

Figure 4: Implementation of 'Optimization Function' in Python

5. Constraints

Now that we have defined the decision variables and the objective function, we must impose different constraints on the model in order to best respect the reality of the problem.

- **Equality constraints**

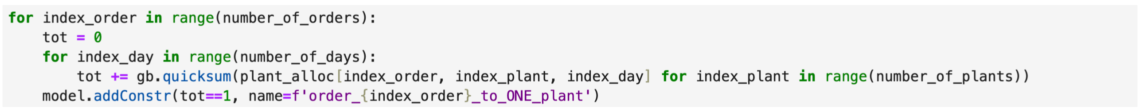
The equality constraints in this model ensure that each order is assigned to one plant for production, one port for shipping, and that specific requirements, like VMI customer needs and product compatibility, are met.

a. Order Plant Allocation

This constraint ensures that each order is assigned to exactly one plant for production. Using the production decision variable $x_{ijd} \epsilon (0,1)^{N \times M \times D}$,where N is the number of orders, M is the number of plants and D is the number of days, the constraint is:

$$\sum_{d=1}^{D} \sum_{j=1}^{M} x_{ijd} = 1 , \forall\, i \in (1, N)$$

I.e. for each order i, the sum of assignments across all plants j and all days d must be equal to 1, making sure that each order is produced at only one plant on one day.



```python
for index_order in range(number_of_orders):
    tot = 0
    for index_day in range(number_of_days):
        tot += gb.quicksum(plant_alloc[index_order, index_plant, index_day] for index_plant in range(number_of_plants))
    model.addConstr(tot==1, name=f'order_{index_order}_to_ONE_plant')
```

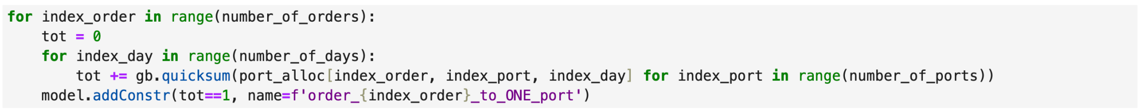
Figure 5: Implementation of the constraint 'Order Plant Allocation' in Python

b. Order Port Allocation

Similarly, this constraint makes sure that each order is shipped through exactly one port. Using the shipping decision variable $y_{ikd} \epsilon (0,1)^{N \times K \times D}$, where K now is the number of ports, the constraint is:

$$\sum_{d=1}^{D} \sum_{k=1}^{K} y_{ikd} = 1 , \qquad \forall\, i \in (1, N)$$

This means that each order i is assigned to only one port for shipping over all days.



```python
for index_order in range(number_of_orders):
    tot = 0
    for index_day in range(number_of_days):
        tot += gb.quicksum(port_alloc[index_order, index_port, index_day] for index_port in range(number_of_ports))
    model.addConstr(tot==1, name=f'order_{index_order}_to_ONE_port')
```

Figure 6: Implementation of the constraint 'Order Port Allocation' in Python

c. VMI Customers

Some orders are issued by Vendor-Managed Inventory (VMI) customers, meaning they must be supplied by specific plants. This is a conditional constraint that only applies if the customer of an order is a VMI customer. We can check if an order's customer is in the list of VMI customers and then find the specific plants that can supply this customer.

It is more elegant to "invert" the logic for this condition and check if a *plant* has a VMI agreement with an order's *customer*. Because one customer can have VMI agreements with more than one plant the allocation variables of only the plants that do *not* have a VMI agreement with that customer have to be constrained to zero. This ensures that the optimization algorithm can still choose between all plants that have a VMI agreement with the customer.

We can state this constraint more specifically using mathematical notation. For all orders i, let Q_i be the set of plants that have a VMI agreement with the customer C_i of order i. If Q_i is not empty we add the following constraint for the plant allocation variables $x_{i,j,d}$. Here d denotes the index of the day at which the order is allocated.

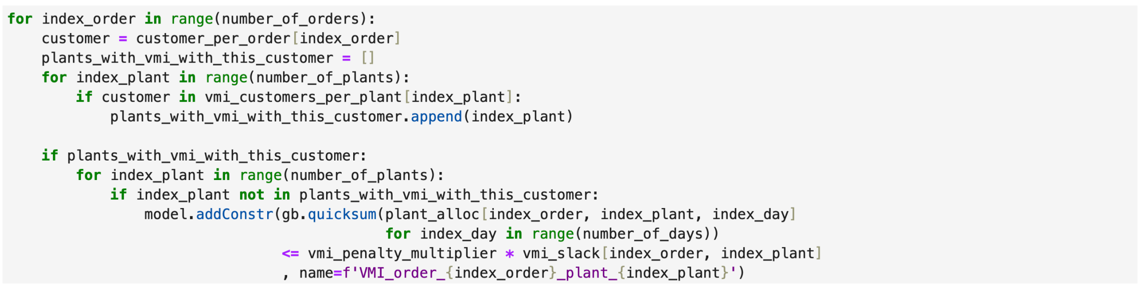
$$\sum_{d=1}^{D} x_{i,j,d} = 0 \ , \forall\, j \in (1, M) \backslash Q_i$$

The dataset contains discrepancies that conflict with this constraint. For example some customers place orders with products that cannot be produced by the plants they have a VMI agreement with. Because of these discrepancies it became necessary to transform the equality constraint into an inequality constraint with a slack variable for the VMI constraint vs. This slack variable allows the optimization algorithm to violate the constraint, i.e. allocate the order of a customer with a VMI agreement with a plant that cannot produce its order to another plant the customer has no VMI agreement with.

The transformed constraint that only applies under the conditions specified earlier is:

$$\sum_{d=1}^{D} x_{i,j,d} \leq P \cdot vs_{i,j} \ , \forall\, j \in (1, M) \backslash Q_i$$

Figure 7 shows the implementation of this constraint in python.



```python
for index_order in range(number_of_orders):
    customer = customer_per_order[index_order]
    plants_with_vmi_with_this_customer = []
    for index_plant in range(number_of_plants):
        if customer in vmi_customers_per_plant[index_plant]:
            plants_with_vmi_with_this_customer.append(index_plant)

    if plants_with_vmi_with_this_customer:
        for index_plant in range(number_of_plants):
            if index_plant not in plants_with_vmi_with_this_customer:
                model.addConstr(gb.quicksum(plant_alloc[index_order, index_plant, index_day]
                                            for index_day in range(number_of_days))
                                <= vmi_penalty_multiplier * vmi_slack[index_order, index_plant]
                                , name=f'VMI_order_{index_order}_plant_{index_plant}')
```

Figure 7: Implementation of the constraint 'VMI Customers' in Python

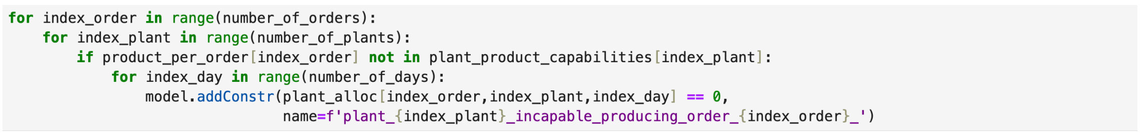
d. Plant Product Capabilities

Not all plants can produce all products. Therefore, orders with products that a plant cannot produce should not be assigned to that plant. Since each order contains only one product type, we can check the product ID of each order and the plants that cannot handle this product.

With x_{ijd} as the plant allocation variable, $OPID_i$ as the product ID of order i, and $PPID_j$ as the set of product IDs that plant j can produce, the constraint is:

$$\sum_{d=1}^{D} x_{ijd} = 0 \ if \ OPID_i \notin \ PPID_j, \qquad \forall\, i \in (1, N), j \in (1, M)$$

This ensures that orders are only assigned to plants that can produce the requested product.



```
for index_order in range(number_of_orders):
    for index_plant in range(number_of_plants):
        if product_per_order[index_order] not in plant_product_capabilities[index_plant]:
            for index_day in range(number_of_days):
                model.addConstr(plant_alloc[index_order,index_plant,index_day] == 0,
                                name=f'plant_{index_plant}_incapable_producing_order_{index_order}_')
```

Figure 8: Implementation of the constraint 'Plant Product Capabilities' in Python

- **Inequality constraints**

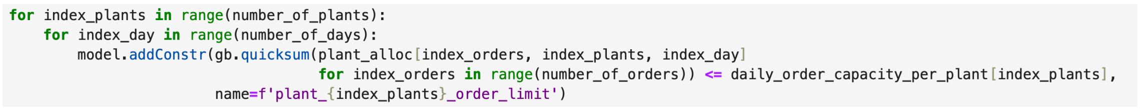
Along with the equality constraints, this model requires inequality constraints to handle plant capacity limits and plant-port connections.

a. Plant Capacity Limit

Each plant has a daily capacity limit on the number of orders it can process. We can calculate the number of orders assigned to each plant by summing the plant allocation variable x_{ijd} for all orders assigned to that plant. Using DC_j as the daily capacity for plant j, we write this constraint as:

$$\sum_{i=1}^{N} x_{ijd} \leq DC_j, \qquad \forall\, i \in (1, M)\,, d \in (1, D)$$

This ensures that each plant j does not exceed its maximum number of orders per day for all days.



```
for index_plants in range(number_of_plants):
    for index_day in range(number_of_days):
        model.addConstr(gb.quicksum(plant_alloc[index_orders, index_plants, index_day]
                                    for index_orders in range(number_of_orders)) <= daily_order_capacity_per_plant[index_plants],
                        name=f'plant_{index_plants}_order_limit')
```

Figure 9: Implementation of the constraint 'Plant Capacity Limit' in Python

b. Plant Port Connections

In this problem, only some ports can be reached by the plants. We can formulate this as an inequality constraint. For each order and each plant, we check which ports are connected to the plant. If a port is not connected to the plant, we make sure that the port allocation variable and the plant allocation variable are not both set to 1 at the same time. If they were both 1, it would mean that this order would be shipped via a port that is not connected to the assigned plant.

Using the plant allocation variable x_{ij} for order i and plant j, the port allocation variable y_{ik} for order i and port k, and the connected ports CP_j for plant j, we can write:

$$\sum_{d=1}^{D} x_{ijd} + y_{ikd} \leq 1 \; if \; k \notin CP_j, \qquad \forall\, i \in (0, N-1), j \in (0, M-1), k \in (0, K-1)$$

This ensures that an order is only shipped via a port that is connected to the plant it was assigned to.

```python
for index_order in range(number_of_orders):
    for index_plant in range(number_of_plants):
        for index_port in range(number_of_ports):
            if index_port not in ports_per_plant[index_plant]:
                tot = 0
                for index_day in range(number_of_days):
                    tot += plant_alloc[index_order, index_plant, index_day] + port_alloc[index_order, index_port, index_day]
                model.addConstr(tot <= 1, name=f'order_{index_order}_plant_{index_plant}_not_conn_to_port_{index_port}')
```

Figure 10: Implementation of the constraint 'Plant Port Connection' in Python

6. Results

Under the described constraints we get a solution with an objective function value of $1.0452 \cdot 10^{12}$ \$. In the solution 1045 VMI constraints get violated. After deducting the penalty from the objective function the optimal cost found with this model evaluates to $16.18\, Mio.\$$.

Figure 11 illustrates the order allocation using a Sankey diagram. It quickly becomes apparent that plant three is the backbone of the allocation operation. It seems to process roughly half of all orders. All other plants are significantly less busy.

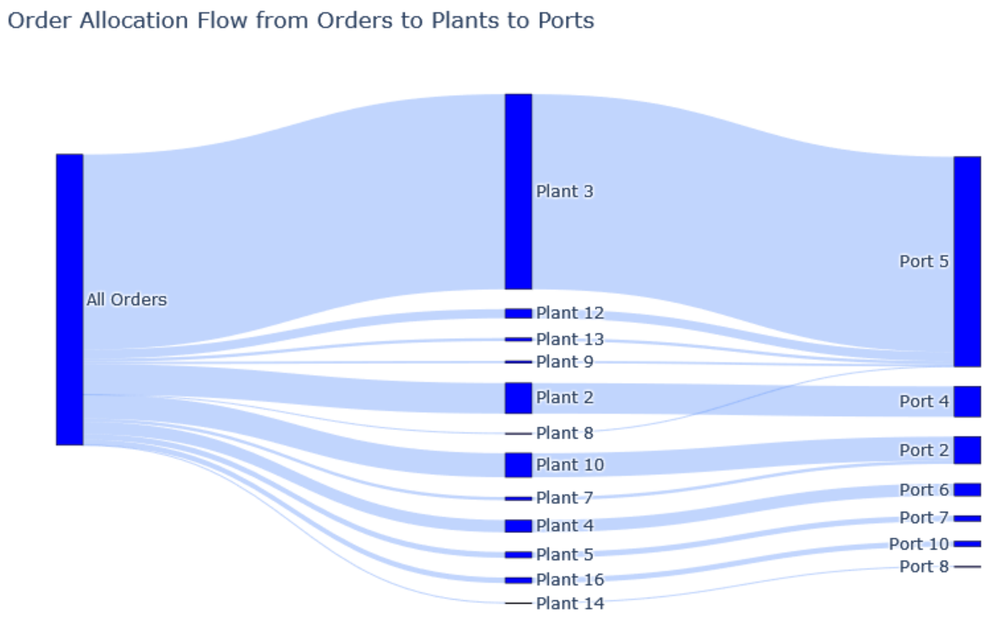


Figure 11: Sankey diagram of order allocation

Port five sems to use its connections to multiple plants well and processes more than half of the orders. Port two and four also contribute significantly while the other ports contribute less to shipping the orders.

In Figure 12 the violating orders are highlighted in a Sankey diagram similar to the one in Figure 11. Apparently the majority of the orders are allocated without violating the VMI constraints. Of the violating orders most get allocated to plant three.

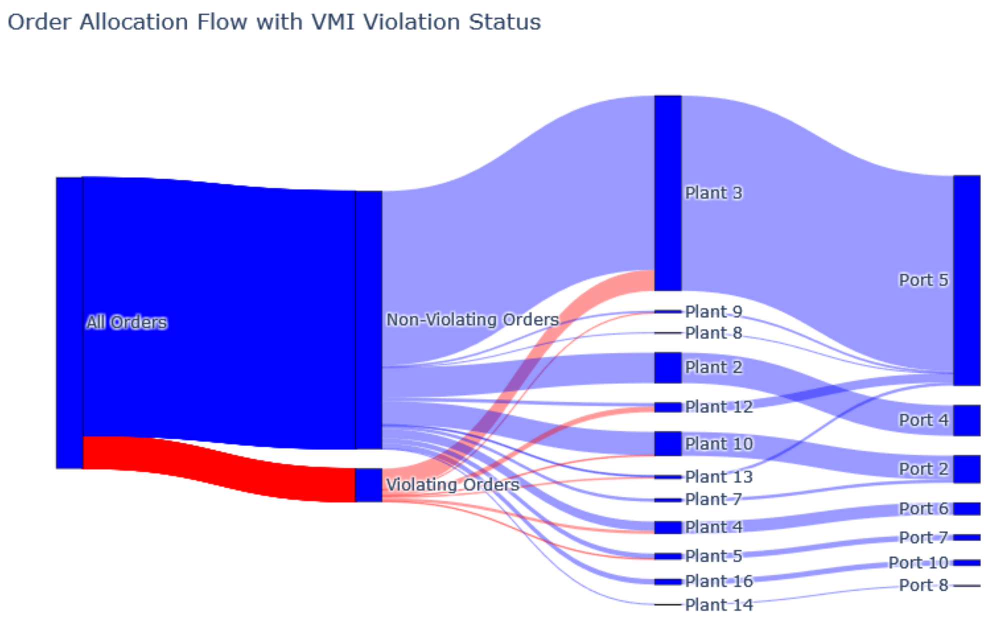


Figure 12: Sankey diagram of order allocation with VMI violations highlighted

Conclusion

In conclusion, this project was difficult and sometimes confusing, especially as we had to choose the topic ourselves. However, the experience helped us to progress, as an engineer must be able to solve problems that have sometimes never been solved before. Sometimes you find a feasible and satisfactory solution, sometimes you don't. The lesson learned is that without effort, no solution will be found. This project also highlighted the power and flexibility of MIP programming for complex optimization tasks.

The difficulty with this model was that it involved solving both optimization problems within the same project. Working with a real data set was interesting, but also challenging, as it required several adaptations, demonstrating the importance of data processing in any IT project. The nature of the data highlighted the fact that careful data processing is essential for optimization.

We may have run out of time to go further, and had to make a few assumptions and simplifications. Nevertheless, we feel that this project provides a legitimate vision of the optimization we wanted to achieve. This experience has given us valuable insight into real-world optimization and the skills needed to deal with complex logistics problems.

References

[1] Kalganova, Tatiana, & Dzalbs, Ilja. (2019). *Supply Chain Logistics Problem Dataset*. Brunel University London, 2019. Available: https://brunel.figshare.com/articles/dataset/Supply_Chain_Logistics_Problem_Dataset/7558679?file=20162015