ISyE 6669 Project - Team 7 - Part A - Fall 2020

Connor Owen, Matthew Mendez & Peter Williams

date: 2020-11-18

Introduction

You are in charge of developing an optimization solution for a large online retailer. The retailer sells K types of products. These products are stored in N warehouses spread around the country. Each warehouse has limited stock of products available. These are given in Warehouses.csv. We want to satisfy M customer orders. Each order may consist of varying quantities of different products. For now, you can assume that products are infinitely divisible.

For instance, all of the following are valid orders:

- *Order A requires 3 units of Product 1.*
- *Order B requires 5 units of Product 2 and 5 units of Product 3*
- *Order C requires 1.5 unit of Product 1*

Additionally, an order can be fulfilled from multiple warehouses. For example, if an order requires 2 units of Product 1 and 3 units of Product 2, you could send:

- *1 unit of Product 1 from Warehouse 1.*
- *1 unit of Product 1 and 1.5 units of Product 2 from Warehouse 2*
- *1.5 units of Product 2 from Warehouse 3*

The orders are given in Orders.csv.

Sending products from the warehouse to the customer has a cost. The cost is proportional to both the distance between the warehouse and the customer and the total weight of the items sent. The costs of sending one pound from a warehouse to satisfy a order are given in DeliveryCost.csv. The per-unit weight of the products is given in ProductWeight.csv.

Your goal is to assign customer orders to warehouses so that you satisfy all the orders while minimizing fulfillment costs.

Part A (50 pts)

Question 1 (20 pts)

Formulate a Linear Programming model to solve this problem. Your submission must be typed. Clearly define all variables, parameters, and constraints.

To begin we list products, warehouses, and orders according to the following notation:

$$k = 1, ..., K \text{ Products}, \tag{1}$$
$$j = 1, ..., N \text{ Warehouses, and} \tag{2}$$
$$i = 1, ..., H \text{ Orders} \tag{3}$$

We then define,

$$d_{ik} \text{ , the demand for product } k \text{ in order } i, \tag{4}$$
$$s_{jk} \text{ , stock of product } k \text{ in warehouse } j, \tag{5}$$
$$c_{ij} \text{ , cost of sending 1 lbs. of product from warehouse } j \text{ to customer } i, \tag{6}$$
$$w_k, (k = 1, ..., K) \text{ denotes the weight of product k in lbs.} \tag{7}$$

Our decision variables are formulated as,

$$x_{ijk}, \text{ number of units shipped of product } k \text{ from warehouse } j \text{ to customer } i, \text{ and} \tag{8}$$
$$\delta^*_{ik} = \text{ the number of units of product } k \text{ not fulfilled in order } i \tag{9}$$

and our objective is then to minimize the cost of shipping units to customers:

$$\min z = \sum_{i=1}^{H}\sum_{j=1}^{N}\sum_{k=1}^{K} x_{ijk}c_{ij}w_k + M\sum_{i=1}^{H}\sum_{k=1}^{K}\delta^*_{ik} \tag{10}$$

subject to,

$$\sum_{j=1}^{N} x_{ijk} + \delta^*_{ik} = d_{ik}, \forall i \in M, k \in K, \tag{11}$$
$$\sum_{i=1}^{H} x_{ijk} \leq s_{jk}, \forall j \in N, k \in K \tag{12}$$
$$x_{ijk} \geq 0 \forall i \in H, j \in N, k \in K \tag{13}$$
$$\delta^*_{ik} \geq 0 \ \forall i \in H, k \in K \tag{14}$$
$$M > 0, \text{ arbitrary large positive number (i.e. 'Big M').} \tag{15}$$

where (11) ensures that all demand must be satisfied or accounted for, (12) ensures that stock levels are not exceeded, and (13) and (14) ensure that non-negativity is enforced.

Question 2 (20 pts)

Implement your model in Xpress or Gurobi/Python. Your program must read the data from the given files. Hard-coded data will be (severely) penalized. In your submission, this script should be named ModelA.mos or ModelA.py. Clearly explain your data structures and how different parts of your code correspond to different parts of your formulation. Specifically, explain which parts of the code generate the variables, objective function, and constraints. Your program's output should clearly show the optimal solution in an easy-to-read way.

We used *Gurobi/Python* to build our model, and our chosen data structures implemented in *ModelA.py* are derived from the imported dataset files. This allows our model to be run for different input files assuming that data input formats remain consistent. The python script relies on the following input data structures:

- The *orders.csv*, a file that is read into our environment as a *DataFrame* type object using the python *pandas* module. We denote this object as *orders_df* in our code. Based on the provided file, this object, in terms of rows and columns, is a 118×3 tabular object consisting of *Order ID*, *Product ID*, and *Quantity* containing information about demand for each product type. By aggregating the sum of *Quantity* along *Order ID* $(i = 1, ..., M)$ and *Product ID* $(k = 1, ..., K)$ we are able to formulate demand corresponding to our model where, $d_{ik} \in \mathbb{R}$, the demand for product k in order i.
- The *ProductWeight.csv* is read into our environment as a *DataFrame* type object denoted as *product_weight_df* in our code. In terms of rows and columns, *product_weight_df* is a 5×1 tabular object consisting of *Weight* for each product type. This allows us to implement the weight component of our formulation, that is, the data in this file provides $w_k \in \mathbb{R}, (k = 1, ..., K)$ the weight of product k in lbs.
- The *Warehouses.csv* is read into our environment as a *DataFrame* type object denoted as *warehouse_df* in our code. In terms of rows and columns, *warehouse_df* is a 40×3 tabular object consisting of *Warehouse ID*, *Product ID*, and *Stock*, i.e. the stock of each product in each warehouse. This allows us to implement the stock constraints in our formulation, that is, the data in this file provides $s_{jk} \in \mathbb{R}$, stock of product k in warehouse j.
- Lastly in our code, the *DeliveryCost.csv* is read into our environment as a *DataFrame* type object denoted as *costs_df* in our code. In terms of rows and columns, *costs_df* is a 8×46 tabular object consisting of *Warehouse ID* in the rows, and the associated cost of an *Order ID* in the columns. To simplify subsequent coding, we re-encode this data into a dictionary object denoted as *costs* which associates the cost of each order shipping from a warehouse. This data supports the formulation of $c_{ij} \in \mathbb{R}$, cost of sending 1 lbs. of product from warehouse j to customer i, since each *Order ID* corresponds to a customer.

With these input data objects, we create indices over which our code iterates over warehouses, orders and products to formulate demand, and stock constraints. However, we must deal with the complexity of not being able to fulfill all orders demanded based on stock levels. We solved for this using *Big M* notation above in our model, penalizing unmet demand, denoted in our model as $\delta^*_{ik} \geq 0 \ \forall i \in H, k \in K$. In our code, to account for this we add variables that account from the flow from a warehouse, or a product for an order. We then add constraints to our model that account for flow exceeding stock levels, and an overall constraint to accomodate total order δ. We set $M = 10,000$ in our script implementation to account for the total cost penalty, $M \sum_{i=1}^{H} \sum_{k=1}^{K} \delta^*_{ik}$.

Note that for the purposed of *Gurobi* implementation, all variables in this initial formulation have been encoded as continuous type variables. To illustrate how we built models variables into our ModelA.py script Code that generates variables, we rely on lists indexed for each order and supply point, along leftover supply and unfulfilled demand points as follows:

```
order_delta = m.addVars(order_delta_indices, lb=0, vtype=GRB.CONTINUOUS)
supply_delta = m.addVars(supply_delta_indices, lb=0, vtype=GRB.CONTINUOUS)
tot_unfulfilled_demand = m.addVar(lb=0, vtype=GRB.CONTINUOUS)
tot_leftover_supply = m.addVar(lb=0, vtype=GRB.CONTINUOUS)
```

In order to generates constraints dynamically, we define costs based on weights, and then loop over orders and warehouse to ensure we meet demand and do not exceed stock as the following code block. It also defines our objective penalized by unfulfilled demand:

```
# add costs
m.addConstr(cost == sum(flow[t]*costs[t[0]][t[1]]*product_weights[t[2]] for t in indices))
# meet demand
for o in orders:
    for k in orders_data[o]:
        m.addConstr(sum(flow[(w, o, k)]
          for w in warehouses) + order_delta[(o, k)] == orders_data[o][k])
# don't exceed stock
for w in warehouses:
    for k in products:
        m.addConstr(sum(flow[(w, o, k)]
          for o in orders if k in orders_data[o]) + supply_delta[(w,k)]
                          == warehouse_stock[w][k])

# define order delta total
m.addConstr(tot_unfulfilled_demand == sum(order_delta[t] for t in order_delta_indices))
# define leftover supply total
m.addConstr(tot_leftover_supply == sum(supply_delta[t] for t in supply_delta_indices))

# set new objective
m.setObjective(cost + 10000 * (tot_unfulfilled_demand+tot_leftover_supply),
  sense=GRB.MINIMIZE)
```

Question 3 (5 pts)

Solve your model. What is the objective function value of your solution? What does it mean in words? What is the optimal solution? Make sure to specify which orders are satisfied from which warehouse and what quantities of different items have been sent. In your write up, summarize your solution in a human readable format, e.g. a table.

Based on our implementation in *Gurobi* in *ModelA.py* as described above our realized objective function value for minimization is $z = 165497.8112$ *(rounded)*. However, because we used a *'Big M'* model formulation to account for 16.0 unfulfilled orders with $M = 10,000$ our total order cost is in fact, $165497.8112 - 160000 = 5497.8112$

In terms of our decision variables and output, our model provides two key outputs, the quantity of products planned for fulfillment from each warehouse, for each product *(Table 1)*. It also provides the unfulfilled demand quantity for each product and order *(Table 2)*.

The following provides a preview of both of these tables with actual results included,

Table 1: Preview: Minimizing Order Fulfillment Costs

Order ID	Warehouse ID	Product ID	Quantity Fulfilled
1	3	2	2.0
1	4	2	3.0
.	.	.	.
.	.	.	.
45	8	3	1.0
45	8	4	1.0

Note that the full result for part *A* can be found in the appendix for reference.

Our model output also provides a summary, as mentioned of unfulfilled orders which is provided in total here:

Table 2: Preview: Unfulfilled Orders Results

Order ID	Product ID	Quantity Unfulfilled
17	5	1.0
20	1	3.0
20	3	2.0
31	1	2.0
31	3	3.0
31	5	3.0
39	5	1.0
43	5	1.0

Note that based on the results from our model, the total unfulfilled order demand quantity is 16.0 products.

Question 4 (5 pts)

Notice that all orders in the data require whole quantities of products. In practice, the products are not infinitely divisible. Suppose we were to impose this restriction (you do not have to implement it or change the formulation yet). Would the optimal solution change? If your answer is yes, would the optimal objective value be higher, lower, or stay the same? Why? If your answer is no, why not?

Normally when we are working on a minimization problem imposing the restriction of product quantity always taking integer values, our optimal objective could be higher or stay the same. However, the general formulation of our problem above is similar to the form of a transportation problem. That is because we have:

- A supply or stock of product k in warehouse j yielding our supply points,
- and have i demand points for each customer order, and the demand takes on integer values.
- We also have a variable cost to ship product k from warehouse j to customer i, and our key decision variable is,
- x_{ijk}, number of units shipped of product k from warehouse j to customer i.

As a result, imposing this restriction does not change our solution in our problem.

Appendix

Complete solution, problem A including warehouse, product and order assignments and quantity:

Table 3: Full Solution Problem A: Minimizing Order Fulfillment Costs (1/3)

Order ID	Warehouse ID	Product ID	Quantity Fulfilled
1	3	2	2
1	4	2	3
2	4	3	4
2	4	4	1
3	2	5	2
3	3	2	3
3	3	3	2
4	2	1	1
4	3	1	1
4	8	3	2
5	1	3	1
5	1	5	2
6	8	5	3
7	4	2	2
7	6	2	3
8	5	5	1
8	7	2	1
9	1	1	4
9	1	3	1
9	1	5	2
10	1	3	3
10	2	2	1
10	2	3	5
10	2	4	1
11	7	2	4
12	4	1	3
12	4	5	3
13	5	5	4
14	4	4	2
14	5	1	2
14	5	4	2
14	6	1	1
15	6	3	1
15	7	5	3
16	3	2	1
16	4	4	4
17	3	5	5
17	5	5	1
17	6	1	1
17	6	3	4
17	6	4	3
18	3	1	3
18	3	5	1
19	6	4	3
20	7	1	1
20	7	3	1

Table 4: Full Solution Problem A: Minimizing Order Fulfillment Costs (2/3)

Order ID	Warehouse ID	Product ID	Quantity Fulfilled
21	8	1	2
21	8	4	1
22	4	3	5
22	8	3	1
23	8	1	4
23	8	3	1
23	8	5	4
24	5	1	2
24	5	2	2
24	7	2	3
25	6	2	2
25	6	3	3
25	6	5	2
26	7	1	3
26	7	4	5
26	7	5	3
27	2	1	2
27	2	5	2
27	4	1	2
27	4	5	1
28	2	3	5
29	8	4	4
29	8	5	1
30	2	1	3
30	2	2	4
31	1	1	1
32	3	1	2
32	3	3	2
32	5	3	1
33	1	4	6
33	8	3	3
34	5	5	1
34	6	2	3
34	6	3	1
34	6	5	3
35	3	1	3
36	5	1	5
36	5	2	4
36	5	4	4
37	3	3	2
37	3	5	4
38	2	4	4
38	5	2	4
38	5	4	1
39	1	2	3
39	1	5	3
40	6	1	3
40	8	2	2

Table 5: Full Solution Problem A: Minimizing Order Fulfillment Costs (3/3)

Order ID	Warehouse ID	Product ID	Quantity Fulfilled
41	4	1	3
41	4	3	1
41	4	5	1
41	7	1	1
41	7	3	6
41	7	5	3
42	4	4	2
43	2	4	4
43	2	5	3
43	5	3	1
43	6	2	2
43	6	3	1
44	1	2	2
44	1	5	2
44	3	2	2
44	5	3	4
45	8	1	1
45	8	2	4
45	8	3	1
45	8	4	1

Table 6: Problem A: Unfulfilled Orders Results (1/1)

Order ID	Product ID	Quantity Unfulfilled
17	5	1.0
20	1	3.0
20	3	2.0
31	1	2.0
31	3	3.0
31	5	3.0
39	5	1.0
43	5	1.0