Building a Multimode, Multiproduct Supply Chain Optimization model using Python

A Report submitted in fulfilment of the requirements for the Internship Project under the guidance of Dr. Subhas C. Misra

Done By,

Rahul N

4th Year B. Tech, Department of Chemical Engineering

National Institute of Technology, Andhra Pradesh

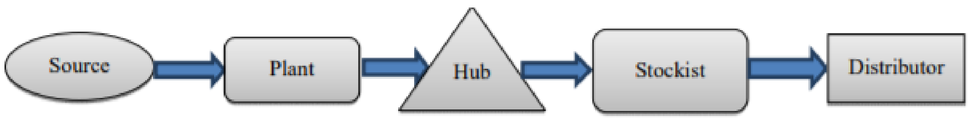
Summary

This Project aims to optimize the material flow in a multi-mode, multi – echelon, multi – product, multi – mode supply chain. To carry out this optimization, we consider 3 stages.

1. Stage 1: Finding the Initial Basic Feasible Solution of the total distribution cost for a given supply and demand using Vogel's Approximation Method.
2. Stage 2: Optimizing the Basic Feasible Solution to find the optimal solution of the distribution among the network using the Method of Multipliers and calculating the distribution cost associated with this solution.
3. Stage 3: Calculating the costs associated with the Vehicle Scheduling problem considering different modes of transportation.

Introduction to the problem and Stage 1&2

We consider a four-stage supply chain network as shown. Our goal is to optimize the material flow for a number of products being transported using a variety of modes of transportation such as the railways or trucks and so on.



Each layer in this network has its own supply and demand (excluding the source). In this project, I have considered only a balanced supply chain problem due the simplicity in calculations and the lack of availability of datasets. Therefore, I have generated a balanced supply and demand and the distribution cost associated at each echelon. However, the problem can be further extended towards an unbalanced supply chain problem with the addition of penalties in the matrix. Our goal is to minimize this distribution cost for each product while satisfying the supply and demand at each stage.

```python
#Functions that generates balanced supply and demand with distribution costs.

def GenerateDistributionCosts(n_supply, n_demand):
    import numpy as np
    return np.random.randint(1, 10, n_supply*n_demand)

def RandomArrayWithConstSum(m, n):
    arr = [0] * m;
    for i in range(n) :
        arr[np.random.randint(0, n) % m] += 1;
    return np.asarray(arr)

def GenerateSupplyAndDemand(n_supply, n_demand):
    supply = np.random.randint(100, 300, n_supply)
    demand = RandomArrayWithConstSum(n_demand, np.sum(supply))
    return (supply, demand)
```

Now, we enter into the first stage of the problem where we try to find an initial basic feasible solution for a given stage, which has a given supply, demand and its associated distribution cost

Let us consider the first stage of the supply chain network, where we employ the Vogel's Approximation method to find our initial basic feasible solution. This function takes in the supply, demand and the distribution costs associated with this network for a single product and returns the initial basic feasible solution as demonstrated below.

```python
#Vogel Approximation Method (VAM) to determine basic feasible solution

def VogelApproxMethod(supply, demand, distribution_costs):
    supply1 = supply
    demand1 = demand
    distribution_costs = distribution_costs
    i = 0
    j = 0
    bfs = []
    while len(bfs) < len(supply) + len(demand) - 1:
        s = supply1[i]
        d = demand1[j]
        v = min(s, d)
        supply1[i] -= v
        demand1[j] -= v
        bfs.append(((i, j), v))
        if supply1[i] == 0 and i < len(supply) - 1:
            i += 1
        elif demand1[j] == 0 and j < len(demand) - 1:
            j += 1
    cost = 0
    for item in bfs:
        cost = cost + distribution_costs[item[0][0]][item[0][1]]*item[1]
    return bfs
```

This method can also return the cost associated with the initial basic feasible solution, but it is not necessary for our further steps. One must note that this cost can be used to check if our model has optimized the initial basic feasible solution or not.

We then use the initial basic feasible solution arrived at the end of this method and optimize using the Method of Multipliers as given in the algorithm below. To implement this algorithm, we require certain "Helper Functions" to simplify our task, which has also been illustrated below.

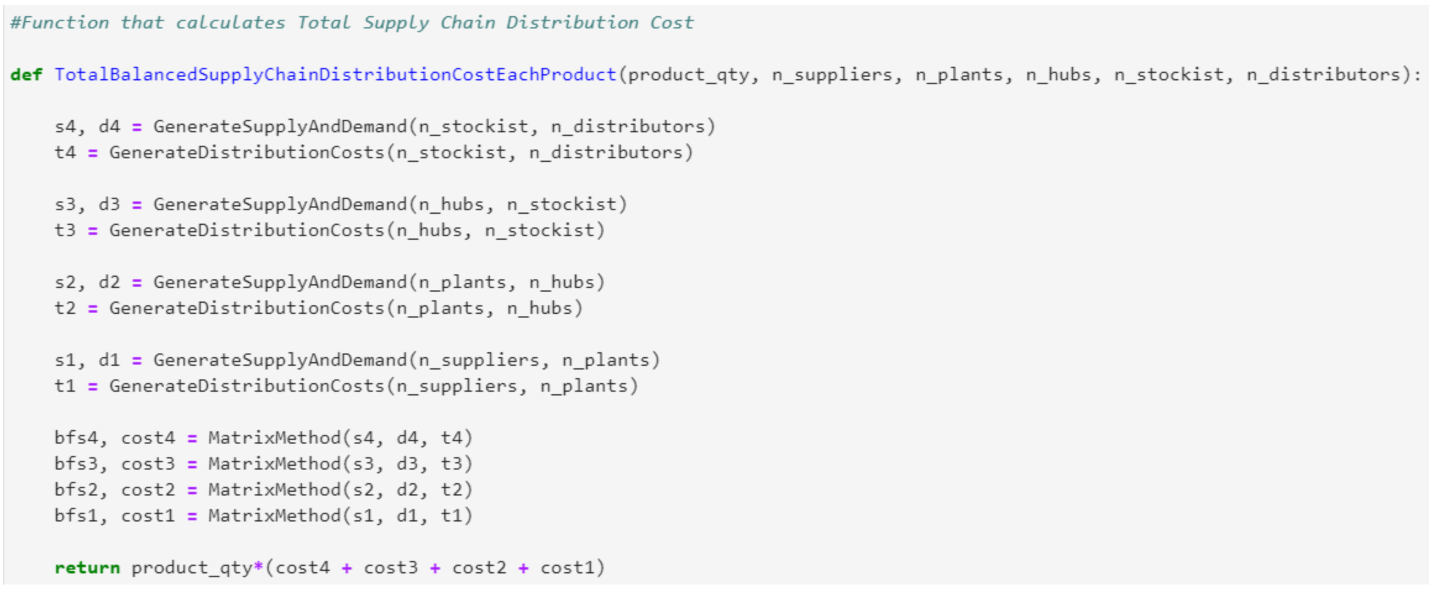
```python
#Recursively trying to find the optimum solution and the cost associated with it using Multiplier Method.

def MatrixMethod(supply, demand, costs):
    costs = costs.reshape(len(supply), len(demand)).tolist()

    def GoInner(bfs):
        u, v = FindUandV(bfs, costs)
        w = FindW(bfs, costs, u, v)
        if ImproveCheck(w):
            ev_position = FindEVPosition(w)
            loop = FindLoop([p for p, v in bfs], ev_position)
            return GoInner(PivotLoop(bfs, loop))
        return bfs

    basic_variables = GoInner(VogelApproxMethod(supply, demand, costs))
    ans = np.zeros((len(costs), len(costs[0])))
    for (i, j), v in basic_variables:
        ans[i][j] = int(v)

    def FindTotalCost(costs, bfs):
        total_cost = 0
        for i, row in enumerate(costs):
            for j, cost in enumerate(row):
                total_cost += cost * bfs[i][j]
        return total_cost
    return ans, FindTotalCost(costs, ans)
```

```python
#Helper Functions to determine optimum solution using Multiplier Method

def FindUandV(bfs, costs):
    u = [None] * len(costs)
    v = [None] * len(costs[0])
    u[0] = 0
    bfs1 = bfs.copy()
    while len(bfs1) > 0:
        for index, bv in enumerate(bfs1):
            i, j = bv[0]
            if u[i] is None and v[j] is None: continue

            cost = costs[i][j]
            if u[i] is None:
                u[i] = cost - v[j]
            else:
                v[j] = cost - u[i]
            bfs1.pop(index)
            break

    return u, v

def FindW(bfs, costs, u, v):
    w = []
    for i, row in enumerate(costs):
        for j, cost in enumerate(row):
            non_basic = all([p[0] != i or p[1] != j for p, v in bfs])
            if non_basic:
                w.append(((i, j), u[i] + v[j] - cost))

    return w
```

```python
def ImproveCheck(w):
    for p, v in w:
        if v > 0: return True
    return False

def FindEVPosition(w):
    w1 = w.copy()
    w1.sort(key=lambda w: w[1])
    return w1[-1][0]

def FindNextNode(loop, not_visited):
    last_node = loop[-1]
    nodes_in_row = [n for n in not_visited if n[0] == last_node[0]]
    nodes_in_column = [n for n in not_visited if n[1] == last_node[1]]
    if len(loop) < 2:
        return nodes_in_row + nodes_in_column
    else:
        prev_node = loop[-2]
        row_move = prev_node[0] == last_node[0]
        if row_move: return nodes_in_column
        return nodes_in_row
```

```python
def FindLoop(bv_positions, ev_position):
    def Inner(loop):
        if len(loop) > 3:
            can_be_closed = len(FindNextNode(loop, [ev_position])) == 1
            if can_be_closed: return loop

        not_visited = list(set(bv_positions) - set(loop))
        possible_next_nodes = FindNextNode(loop, not_visited)
        for next_node in possible_next_nodes:
            new_loop = Inner(loop + [next_node])
            if new_loop: return new_loop

    return Inner([ev_position])

def PivotLoop(bfs, loop):
    even_cells = loop[0::2]
    odd_cells = loop[1::2]
    get_bv = lambda pos: next(v for p, v in bfs if p == pos)
    leaving_position = sorted(odd_cells, key=get_bv)[0]
    leaving_value = get_bv(leaving_position)

    new_bfs = []
    for p, v in [bv for bv in bfs if bv[0] != leaving_position] + [(loop[0], 0)]:
        if p in even_cells:
            v += leaving_value
        elif p in odd_cells:
            v -= leaving_value
        new_bfs.append((p, v))

    return new_bfs
```

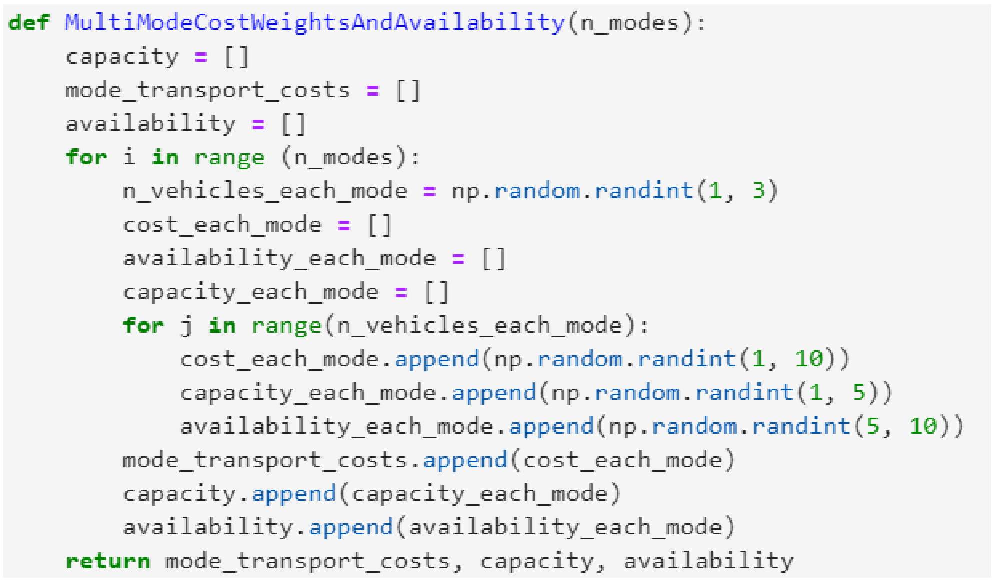
We further calculate the Total Supply Chain Distribution cost after optimization for the supply and demand for all the products with their respective quantities, and we employ the below function to do this task.



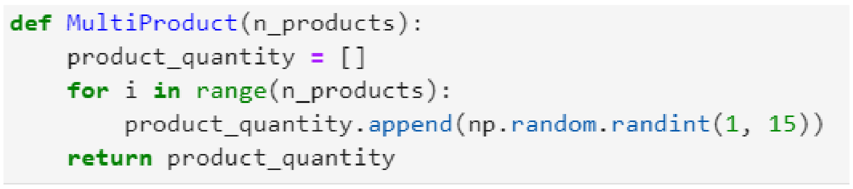
```python
#Function that calculates Total Supply Chain Distribution Cost

def TotalBalancedSupplyChainDistributionCostEachProduct(product_qty, n_suppliers, n_plants, n_hubs, n_stockist, n_distributors):

    s4, d4 = GenerateSupplyAndDemand(n_stockist, n_distributors)
    t4 = GenerateDistributionCosts(n_stockist, n_distributors)

    s3, d3 = GenerateSupplyAndDemand(n_hubs, n_stockist)
    t3 = GenerateDistributionCosts(n_hubs, n_stockist)

    s2, d2 = GenerateSupplyAndDemand(n_plants, n_hubs)
    t2 = GenerateDistributionCosts(n_plants, n_hubs)

    s1, d1 = GenerateSupplyAndDemand(n_suppliers, n_plants)
    t1 = GenerateDistributionCosts(n_suppliers, n_plants)

    bfs4, cost4 = MatrixMethod(s4, d4, t4)
    bfs3, cost3 = MatrixMethod(s3, d3, t3)
    bfs2, cost2 = MatrixMethod(s2, d2, t2)
    bfs1, cost1 = MatrixMethod(s1, d1, t1)

    return product_qty*(cost4 + cost3 + cost2 + cost1)
```

Stage 3: Vehicle Scheduling Problem

We consider a network with the availability of multiple modes of transport. Again, due to simplification of calculations and the unavailability of data, I have generated the Product Quantities, Transport Costs, Capacity of each transportation mode and availability of each mode at each layer using the functions given below.

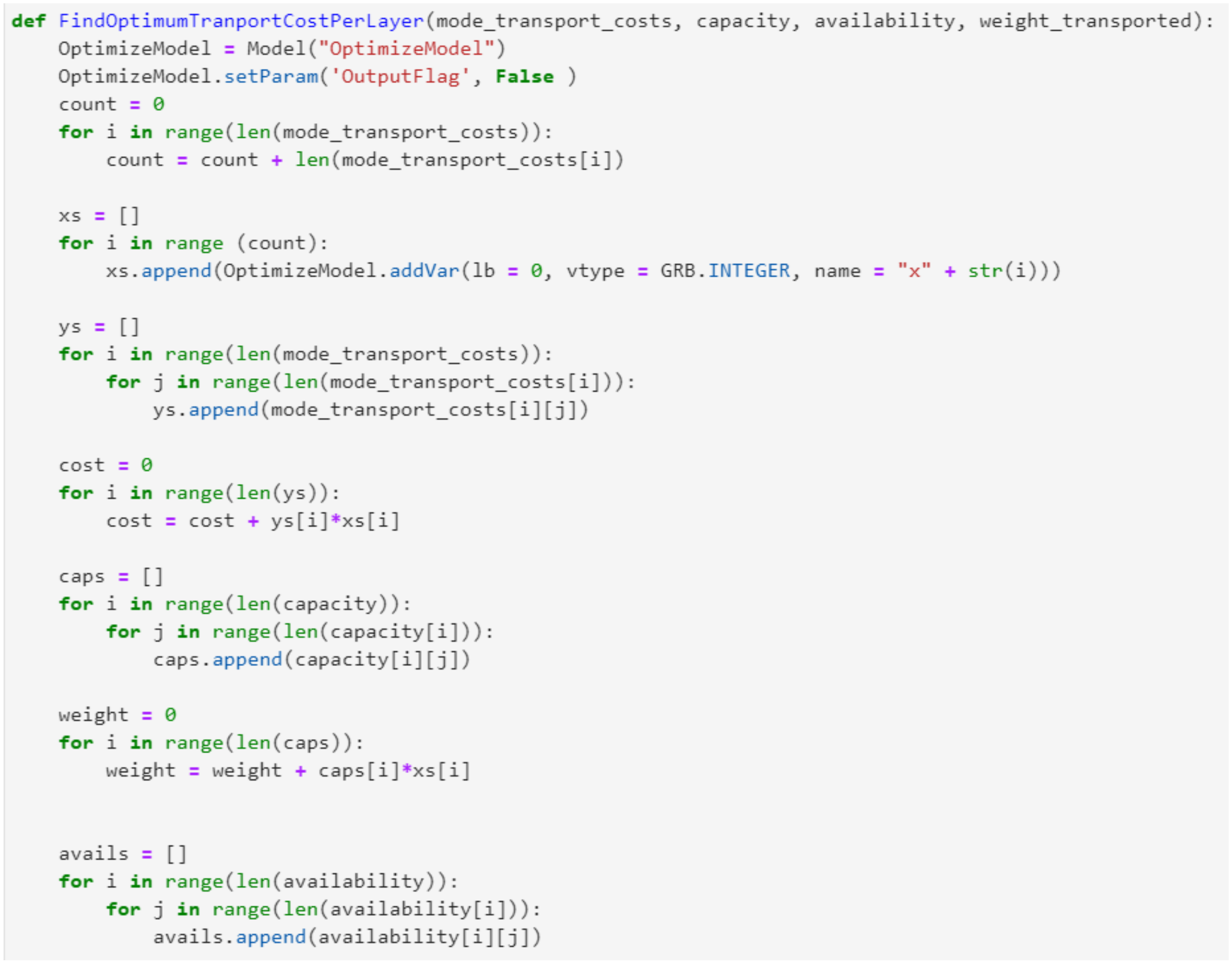


```python
def MultiModeCostWeightsAndAvailability(n_modes):
    capacity = []
    mode_transport_costs = []
    availability = []
    for i in range (n_modes):
        n_vehicles_each_mode = np.random.randint(1, 3)
        cost_each_mode = []
        availability_each_mode = []
        capacity_each_mode = []
        for j in range(n_vehicles_each_mode):
            cost_each_mode.append(np.random.randint(1, 10))
            capacity_each_mode.append(np.random.randint(1, 5))
            availability_each_mode.append(np.random.randint(5, 10))
        mode_transport_costs.append(cost_each_mode)
        capacity.append(capacity_each_mode)
        availability.append(availability_each_mode)
    return mode_transport_costs, capacity, availability
```

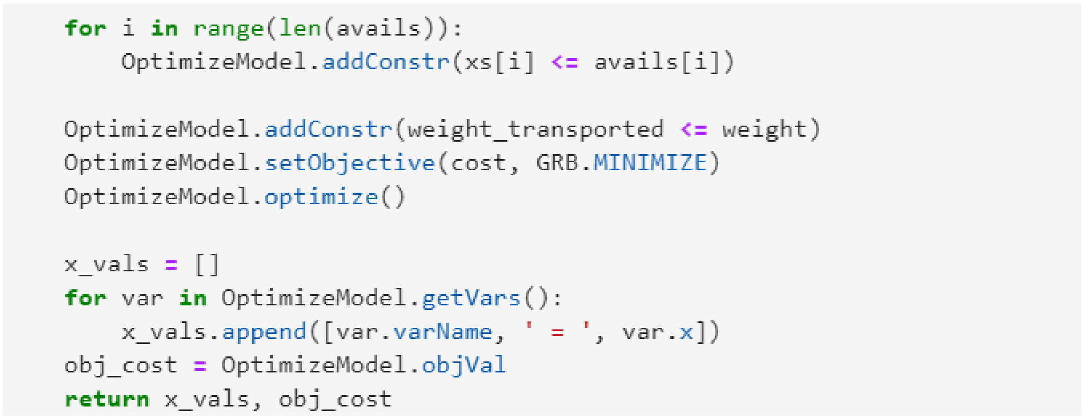


```python
def MultiProduct(n_products):
    product_quantity = []
    for i in range(n_products):
        product_quantity.append(np.random.randint(1, 15))
    return product_quantity
```

To optimize the transport cost per layer, and simplify our calculation time, we make the use of "Gurobi Optimizer" for Python as the VSP reduces to a single-constrained knapsack problem which can be solved easily through this software. The function that calculates this Transportation Cost per layer is given below. We input the transport costs for each mode, and the capacity and availability of each mode and the weight to be transported for that layer, and we get an output of the assignment of different modes of vehicles for each layer and the cost associated with this assignment of vehicles.



```python
def FindOptimumTranportCostPerLayer(mode_transport_costs, capacity, availability, weight_transported):
    OptimizeModel = Model("OptimizeModel")
    OptimizeModel.setParam('OutputFlag', False )
    count = 0
    for i in range(len(mode_transport_costs)):
        count = count + len(mode_transport_costs[i])

    xs = []
    for i in range (count):
        xs.append(OptimizeModel.addVar(lb = 0, vtype = GRB.INTEGER, name = "x" + str(i)))

    ys = []
    for i in range(len(mode_transport_costs)):
        for j in range(len(mode_transport_costs[i])):
            ys.append(mode_transport_costs[i][j])

    cost = 0
    for i in range(len(ys)):
        cost = cost + ys[i]*xs[i]

    caps = []
    for i in range(len(capacity)):
        for j in range(len(capacity[i])):
            caps.append(capacity[i][j])

    weight = 0
    for i in range(len(caps)):
        weight = weight + caps[i]*xs[i]


    avails = []
    for i in range(len(availability)):
        for j in range(len(availability[i])):
            avails.append(availability[i][j])
```



```python
    for i in range(len(avails)):
        OptimizeModel.addConstr(xs[i] <= avails[i])

    OptimizeModel.addConstr(weight_transported <= weight)
    OptimizeModel.setObjective(cost, GRB.MINIMIZE)
    OptimizeModel.optimize()

    x_vals = []
    for var in OptimizeModel.getVars():
        x_vals.append([var.varName, ' = ', var.x])
    obj_cost = OptimizeModel.objVal
    return x_vals, obj_cost
```

Results

We now arrive at the final results of solving this transportation problem. Since we have taken random data, it becomes easy for us to visualize the results using the code below. We have taken the number of products as 2 and 3 and have tabulated our results below.

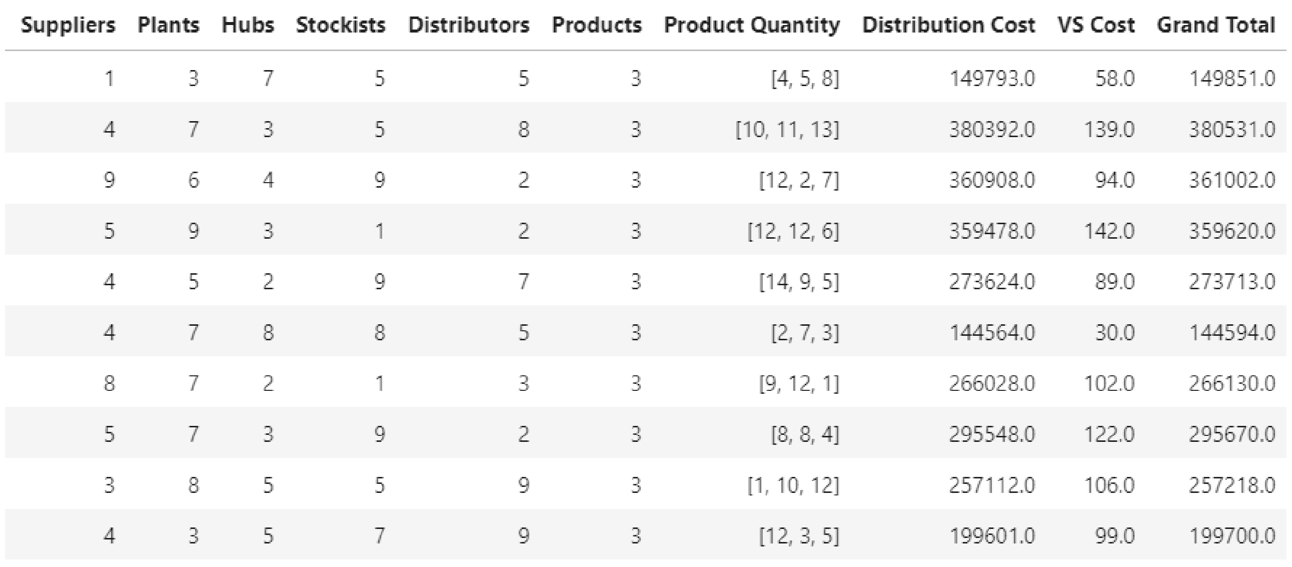
```
#Obtaining Results
```

```
results = []
```

```
for i in range(10):
    n_products = 2
    n_stockist = np.random.randint(1, 10)
    n_suppliers = np.random.randint(1, 10)
    n_plants = np.random.randint(1, 10)
    n_hubs = np.random.randint(1, 10)
    n_distributors = np.random.randint(1, 10)
    product_qty = MultiProduct(n_products)
    total_cost = 0
    vs_cost = 0
    total_weight = 0
    total_vs_cost = 0
    for k in range(4):
        mode_transport_costs, capacity, availability = MultiModeCostWeightsAndAvailability(3)
        weight_transported = np.random.randint(1, 3)*(sum(product_qty))
        total_weight = total_weight + weight_transported
        x_vals, vs_cost = FindOptimumTranportCostPerLayer(mode_transport_costs, capacity, availability, weight_transported)
        total_vs_cost = total_vs_cost + vs_cost
    for j in product_qty:
        total_cost = total_cost + TotalBalancedSupplyChainDistributionCostEachProduct(j, n_suppliers, n_plants, n_hubs, n_stockist, n_distributors)

    results.append([n_suppliers, n_plants, n_hubs, n_stockist, n_distributors, n_products, product_qty, total_cost,
                    total_vs_cost, (total_cost + total_vs_cost)])
```

```
results = pd.DataFrame(results)
results.columns = ('Suppliers', 'Plants', 'Hubs', 'Stockists', 'Distributors', 'Products', 'Product Quantity', 'Distribution Cost',
                   'VS Cost','Grand Total')
```

Suppliers	Plants	Hubs	Stockists	Distributors	Products	Product Quantity	Total Cost	VS Cost	Grand Total
3	9	8	5	1	2	[14, 4]	262570.0	61.0	262631.0
1	8	2	2	8	2	[14, 3]	151247.0	119.0	151366.0
6	9	5	6	6	2	[9, 5]	177045.0	43.0	177088.0
6	4	9	2	1	2	[5, 8]	156866.0	61.0	156927.0
3	4	7	5	5	2	[8, 6]	140086.0	119.0	140205.0
7	8	7	9	8	2	[10, 10]	283390.0	103.0	283493.0
1	4	6	9	1	2	[12, 9]	285210.0	101.0	285311.0
2	6	1	4	3	2	[5, 6]	124271.0	60.0	124331.0
5	4	4	7	6	2	[13, 2]	179627.0	107.0	179734.0
8	1	7	9	3	2	[10, 1]	176683.0	56.0	176739.0



Suppliers	Plants	Hubs	Stockists	Distributors	Products	Product Quantity	Distribution Cost	VS Cost	Grand Total
1	3	7	5	5	3	[4, 5, 8]	149793.0	58.0	149851.0
4	7	3	5	8	3	[10, 11, 13]	380392.0	139.0	380531.0
9	6	4	9	2	3	[12, 2, 7]	360908.0	94.0	361002.0
5	9	3	1	2	3	[12, 12, 6]	359478.0	142.0	359620.0
4	5	2	9	7	3	[14, 9, 5]	273624.0	89.0	273713.0
4	7	8	8	5	3	[2, 7, 3]	144564.0	30.0	144594.0
8	7	2	1	3	3	[9, 12, 1]	266028.0	102.0	266130.0
5	7	3	9	2	3	[8, 8, 4]	295548.0	122.0	295670.0
3	8	5	5	9	3	[1, 10, 12]	257112.0	106.0	257218.0
4	3	5	7	9	3	[12, 3, 5]	199601.0	99.0	199700.0

Discussion and Further Scope

This problem has been aimed to find out the optimized cost required to fulfil the supply and demand in a four-stage multi-mode, multi-product supply chain transportation network. Further, the individual supply chain echelons can be investigated by running the functions at each layer to arrive at conclusion for the mode and type of vehicles to be used, the weights transported and the costs associated with this distribution and transportation.

This project's scope can be further extended to model the behavioural tendencies of the stakeholders at each layer, with the addition of certain constraints to Stage 2&3.

I wholeheartedly thank Dr. Subash C. Misra for giving me this opportunity to learn about the optimization and working of any real – life supply chain network, and I hope that my learnings reflect in this project. It has been a daunting, yet interesting task to explore a field that is relatively new to me and I thank the professor for accepting my request to work on this project.

Kindly note that if there are any mistakes with the logic or problems involved in this project which I have noticed, or if any additions are to be made, I will be sure to rectify them as required.