Decision Analytics: Assignment 2
Linear Programming

Rajbir Bhattacharjee
R00195734

Table of Contents

Task 1: Production Planning

Loading the data (A)

The data is loaded using pandas

```
    def read_csv(self, sheet_name:str)->pd.DataFrame:
        df = pd.read_excel(self.excel_file_name, sheet_name=sheet_name)
        return df
```

Decision variables (B)

The problem can be broken down into two parts:

1. Factory-Supplier-Material relationship
2. Factory-Customer-Product relationship

The two parts can be seen as two tables joined by the factory.
For these two parts, the decision variables are these

1. A 3-dimensional set of real numbered variables. If a supplier S supplies K units of material M to factory F, then var_sfm[S][F][M] = K
2. A 3-dimensional set of real numbered variables. If a factory F supplies K units of product P to customer C, then var_fcp[F][C][P] = K

```
    def create_factory_customer_product_variables(self):
        ret = {}
        for factory in self.factory_names:
            outer = {}
            for customer in self.customer_names:
                inner = {}
                for product in self.product_names:
                    varname = f"factory:{factory}-customer:{customer}" +\
                        f"-product:{product}"
                    variable = self.solver.NumVar(\
                        0, self.solver.infinity(), varname)
                    inner[product] = variable
                outer[customer] = inner
            ret[factory] = outer
        return ret

    def create_supplier_factory_material_variables(self):
        ret = {}
        for supplier in self.supplier_names:
            outer = {}
            for factory in self.factory_names:
                inner = {}
                for material in self.material_names:
                    varname = f"supplier:{supplier}-factory:{factory}" +\
                        f"-material:{material}"
                    variable = self.solver.NumVar(\
                        0, self.solver.infinity(), varname)
                    inner[material] = variable
                outer[factory] = inner
            ret[supplier] = outer
        return ret
```

Constraints with suppliers and factories (E, F)

Constraints are needed to ensure that suppliers do not exceed their stock, and all factories have enough material for the amount of products they produce. This also ties up the two sets of variables (3D collections) together.

```
    # Sheet 4
    # For each factory and material
    # Incoming >= Outgoing
    # OR
    # 0 <= Incoming - Outgoing <= INF
    def create_constraint_meet_factory_requirements(self):
        for factory in self.factory_names:
            for material in self.material_names:
                constraint = self.solver.Constraint(0, self.solver.infinity())

                # Incoming constraints
                for supplier in self.supplier_names:
                    var = self.var_sfm[supplier][factory][material]
                    constraint.SetCoefficient(var, 1.0)

                for customer in self.customer_names:
                    # Outgoing constraints
                    for product in self.product_names:
                        material_per_unit = \
                            get_element(self.product_requirements_df,\
                                        product,\
                                        material)
                        material_per_unit = float(-1 * material_per_unit)
                        var = self.var_fcp[factory][customer][product]
                        constraint.SetCoefficient(var, float(material_per_unit))


    # Sheet 1
    def create_supplier_stock_constraints(self):
        # Create constraints for stocks each supplier has
        # (fixed values as per excel)

        def set_supplier_zero(supplier:str, material:str):
            # If a supplier doesn't have a material as per the excel
            # force it to be zero
            for factory in self.factory_names:
                var = self.var_sfm[supplier][factory][material]
                constraint = self.solver.Constraint(0, 0)
                constraint.SetCoefficient(var, 1.0)

        def set_supplier_capacity(supplier:str, material:str, capacity:int):
            # if a supplier has a x amount of a material as per the excel
            # force it to be that value
            constraint = self.solver.Constraint(0, capacity)
            for factory in self.factory_names:
                var = self.var_sfm[supplier][factory][material]
                constraint.SetCoefficient(var, 1.0)

        for supplier in self.supplier_names:
            for material in self.material_names:
                capacity = get_element(\
                    self.supplier_stock_df, supplier, material)
                if 0 == capacity:
                    set_supplier_zero(supplier, material)
```

```
            else:
                set_supplier_capacity(supplier, material, capacity)
```

Constraints with Factories and Customers (C, D, G)

Constraints are set up to ensure that each factory has enough production to be able to ship to customers as per the requirements.

```
    # Sheet 5
    def create_production_capacity_constraints(self):
        # Create constraints for production capacity for each factory
        # (as per excel sheet, fixed values)

        def set_zero(factory:str, product:str):
            for customer in self.customer_names:
                var = self.var_fcp[factory][customer][product]
                constraint = self.solver.Constraint(0, 0)
                constraint.SetCoefficient(var, 1.0)

        def set_capacity(factory:str, product:str, capacity:int):
            constraint = self.solver.Constraint(0, capacity)
            for customer in self.customer_names:
                var = self.var_fcp[factory][customer][product]
                constraint.SetCoefficient(var, 1.0)

        for factory in self.factory_names:
            for product in self.product_names:
                capacity = get_element(\
                    self.production_capacity_df, product, factory)
                if 0 == capacity:
                    set_zero(factory, product)
                else:
                    set_capacity(factory, product, capacity)


    # Sheet 7
    def create_constraint_meet_customer_demands(self):
        # Get the demand for each customer from the excel sheet
        # Then loop over each factory, and ensure that the total
        # from all factories to that customer meets the demand

        def set_zero(customer:str, product:str):
            for factory in self.factory_names:
                var = self.var_fcp[factory][customer][product]
                constraint = self.solver.Constraint(0, self.solver.infinity())
                constraint.SetCoefficient(var, 1.0)

        def set_demand(customer:str, product:str, demand):
            constraint = self.solver.Constraint(demand, self.solver.infinity())
            for factory in self.factory_names:
                var = self.var_fcp[factory][customer][product]
                constraint.SetCoefficient(var, 1.0)

        for product in self.product_names:
            for customer in self.customer_names:
                demand = get_element(self.customer_demand_df, product, customer)
                if 0 == demand:
```

```
                set_zero(customer, product)
            else:
                set_demand(customer, product, demand)
```

Setting up the objective function (H, I)

The objective is that the cost of all the items produced should be the minimum. The cost can again, be thought of as two parts:

1. Cost of materials to each factory including shipping cost
2. Cost of products to each customer, including shipping cost

For the two parts, again two 3-dimensional sets of coefficients are first calculated

1. coeff_fcp – the coefficients for each variable in var_fcp
2. coeff_sfm – the coefficients for each variable in var_sfm

Each coefficient for var_fcp must be a sum of both material cost and material shipping cost. It is easier to accumulate them in two separate steps, and hence the need for a separate array of coefficients. The case is similar for var_sfm, where there are shipping and production costs involved.

```
    # Sheet 6
    def accumulate_production_cost(self):
        # coeff_fcp is an accumulator of production cost, shipping cost,
        # material cost, etc.
        # In this function, add the production cost for each
        # factory, customer, product
        for factory in self.factory_names:
            for product in self.product_names:
                cost_per_unit = get_element(\
                    self.production_cost_df, product, factory)
                cost_per_unit = float(cost_per_unit)
                if cost_per_unit == float('inf'): cost_per_unit = 0.0
                for customer in self.customer_names:
                    self.coeff_fcp[factory][customer][product] += cost_per_unit

    # Sheet 8
    def accumulate_shipping_cost(self):
        # coeff_fcp is an accumulator of production cost and shipping cost
        # In this function, add the shipping cost for each
        # factory, customer, product
        for factory in self.factory_names:
            for customer in self.customer_names:
                shipping_cost_per_unit = get_element(\
                    self.shipping_cost_df, factory, customer)
                shipping_cost_per_unit = float(shipping_cost_per_unit)
                if shipping_cost_per_unit == float('inf'):
                    shipping_cost_per_unit = 0.0
                for product in self.product_names:
                    self.coeff_fcp[factory][customer][product] += \
                        shipping_cost_per_unit

    # Sheet 2: Raw Materials Cost
    def accumulate_raw_materials_cost(self):
        # coeff_sfm is an accumulator for raw materials cost, and raw materials
```

```
        # shipping cost.
        # In this function accumulate the raw materials cost for each
        # supplier, material and factory
        for supplier in self.supplier_names:
            for material in self.material_names:
                material_cost = get_element(\
                    self.raw_material_cost_df, supplier, material)
                material_cost = float(material_cost)
                if material_cost == float('inf'): material_cost = 0.0
                for factory in self.factory_names:
                    self.coeff_sfm[supplier][factory][material] += \
                        material_cost

    # Sheet 3: Raw Metrials Shipping
    def accumulate_raw_materials_shipping_cost(self):
        # coeff_sfm is an accumulator for raw materials cost, and raw materials
        # shipping cost.
        # In this function accumulate the raw materials shipping cost for each
        # supplier, material and factory
        for supplier in self.supplier_names:
            for factory in self.factory_names:
                shipping_cost = get_element(\
                    self.raw_material_shipping_df, supplier, factory)
                shipping_cost = float(shipping_cost)
                if shipping_cost == float('inf'): shipping_cost = 0.0
                for material in self.material_names:
                    self.coeff_sfm[supplier][factory][material] += shipping_cost
```

Finally the objective coefficients are set as below:

```
    def set_objective_coefficients(self):
        # Total cost is a sum of
        # 1. production cost
        # 2. shipping cost
        # 3. raw materials cost
        # 4. Raw materials shipping cost

        # This loop adds up production cost and shipping cost
        # The two have already been added up and stored in coeff_fcp
        for prod in self.product_names:
            for fact in self.factory_names:
                for cust in self.customer_names:
                    var = self.var_fcp[fact][cust][prod]
                    val = self.coeff_fcp[fact][cust][prod]
                    self.cost_objective.SetCoefficient(var, val)

        # This loop adds up raw materials cost and raw materials shipping cost
        # The two have already been added up and stored in coeff_sfm
        for fact in self.factory_names:
            for supp in self.supplier_names:
                for mat in self.material_names:
                    var = self.var_sfm[supp][fact][mat]
                    val = self.coeff_sfm[supp][fact][mat]
                    self.cost_objective.SetCoefficient(var, val)
```

Finally, the objective is set to a minimization task.

Determining how much material has to be ordered from each supplier (J)

This is easily achieved by iterating the var_sfm array

```
    def print_supplier_factory_material(self):
        print()
        print("Printing Supplier Factory Orders")
        print('*' * len("Printing Supplier Factory Orders"))
        print()
        for fact in self.factory_names:
            print(fact)
            print('-' * len(fact))
            for supp in self.supplier_names:
                out_str = f"    {supp} - "
                for mat in self.material_names:
                    value = self.var_sfm[supp][fact][mat].SolutionValue()
                    if (0.0 != value):
                        out_str = out_str + "    "
                        temp = f"{mat:.15s}: {round(value,2):05.2f}"
                        out_str = out_str + f"{temp:23s}"
                print(out_str)
            print()
```

Output

```
Printing Supplier Factory Orders
********************************

Factory A
---------
    Supplier A -     Material A: 20.00          Material B: 20.00
    Supplier B -     Material A: 19.00          Material B: 04.00
    Supplier C -
    Supplier D -     Material C: 14.00          Material D: 50.00
    Supplier E -

Factory B
---------
    Supplier A -
    Supplier B -     Material A: 06.00          Material B: 34.00
    Supplier C -     Material C: 32.00          Material D: 05.00
    Supplier D -     Material C: 06.00
    Supplier E -     Material A: 04.00

Factory C
---------
    Supplier A -
    Supplier B -     Material B: 06.00
    Supplier C -     Material B: 10.00          Material C: 20.00
Material D: 35.00
    Supplier D -
    Supplier E -     Material A: 25.00          Material D: 40.00
```

Printing Supplier Bill for Each Factory (K)

This is easily achieved by iterating the var_sfm array and aggregating:

```
    def print_supplier_bill_for_each_factory(self):
        print()
        print("Printing supplier bill for factories")
        print('*' * len("Printing supplier bill for factories"))
        print()
        for fact in self.factory_names:
            print(fact)
            print('-' * len(fact))
            for supp in self.supplier_names:
                cost = 0.0
                for mat in self.material_names:
                    qty = self.var_sfm[supp][fact][mat].SolutionValue()
                    mat_cost = get_element(self.raw_material_cost_df, supp, mat)
                    shp_cost = get_element(self.raw_material_shipping_df,\
                                           supp, fact)
                    if qty >= EPSILON:
                        cost = cost + (mat_cost + shp_cost) * qty
                print(f"    - {supp:20s} : {round(cost, 2):10.2f}")
            print()
```

Output

```
Printing supplier bill for factories
************************************

Factory A
---------
    - Supplier A           :    2800.00
    - Supplier B           :    2155.00
    - Supplier C           :       0.00
    - Supplier D           :    6440.00
    - Supplier E           :       0.00

Factory B
---------
    - Supplier A           :       0.00
    - Supplier B           :    3500.00
    - Supplier C           :    8550.00
    - Supplier D           :    1380.00
    - Supplier E           :     260.00

Factory C
---------
    - Supplier A           :       0.00
    - Supplier B           :    1590.00
    - Supplier C           :   12450.00
    - Supplier D           :       0.00
    - Supplier E           :    7325.00
```

Units produced per factory and manufacturing cost (L)

This is easily calculated by iterating the var_fcp array and accumulating:

```
    def print_units_and_cost_per_factory(self):
        print()
        print("Printing production and cost for each factory")
        print('*' * len("Printing production and cost for each factory"))
        print()
        for fact in self.factory_names:
            print(fact)
            print('-' * len(fact))
            tot_cost = 0.0
            for prod in self.product_names:
                prod_qty = 0.0

                for cust in self.customer_names:
                    prod_qty = prod_qty +\
                        self.var_fcp[fact][cust][prod].SolutionValue()
                print(f"        {prod:10s}: {round(prod_qty,2):15.2f}")
                prod_cost = get_element(self.production_cost_df, prod, fact)

                if (prod_cost == float('inf') and prod_qty != 0):
                    # If a factory cannot produce an item, it's production
                    # quantity should actually be zero
                    assert(False)
                # Now that we have verified that factories only produce
                # items they can, we simplify the code by setting the
                # production cost to 0
                if (prod_cost == float('inf')): prod_cost = 0
                tot_cost = tot_cost + prod_qty * prod_cost
            print(f"    Total Manufacturing Cost = {round(tot_cost,2):05.2f}")
            print()
```

Output

```
Printing production and cost for each factory
*********************************************

Factory A
---------
        Product A :            6.00
        Product B :            1.00
        Product C :            0.00
        Product D :            3.00
    Total Manufacturing Cost = 1010.00

Factory B
---------
        Product A :            2.00
        Product B :            1.00
        Product C :            4.00
        Product D :            0.00
    Total Manufacturing Cost = 430.00

Factory C
---------
        Product A :            2.00
        Product B :            0.00
        Product C :            0.00
```

```
    Product D :            5.00
Total Manufacturing Cost = 425.00
```

Number of units shipped from each factory to each customer, and shipping costs (M)

This is easily calculated by iterating the var_fcp array and accumulating the numbers.

```
    def print_customer_factory_units_ship_cost(self):
        # For each customer, determine how many units are being shipped
        # from each factory, also the total shipping cost per customer
        print()
        print("Printing shipments for each customer")
        print('*' * len("Printing shipments for each customer"))
        print()
        for cust in self.customer_names:
            ship_cost = 0.0
            print(cust)
            print('-' * len(cust))
            for prod in self.product_names:
                out_str = ""
                out_str = out_str + f"Product {prod:15s} "
                for fact in self.factory_names:
                    qty = self.var_fcp[fact][cust][prod].SolutionValue()
                    if qty >= EPSILON:
                        ship_cost_unit = get_element(self.shipping_cost_df,\
                                                     fact, cust)
                        ship_cost = ship_cost + (qty * ship_cost_unit)
                        out_str = out_str + f"    {fact:.15s} : "
                        out_str = out_str +f"{round(qty,2):5.2f}    "
                print(out_str if out_str != "" else "\n")
            print()
            print(f"Total Shipping Cost: {round(ship_cost,2):5.2f}")
            print()
        print()
```

Output

```
Printing shipments for each customer
************************************

Customer A
----------
Product Product A          Factory A :  5.00        Factory C :  2.00
Product Product B
Product Product C
Product Product D          Factory C :  1.00

Total Shipping Cost: 280.00

Customer B
----------
Product Product A          Factory A :  1.00        Factory B :  2.00
Product Product B
Product Product C
Product Product D

Total Shipping Cost: 110.00
```

```
Customer C
----------
Product Product A
Product Product B           Factory A :  1.00        Factory B :  1.00
Product Product C
Product Product D           Factory C :  3.00

Total Shipping Cost: 370.00

Customer D
----------
Product Product A
Product Product B
Product Product C           Factory B :  4.00
Product Product D           Factory A :  3.00        Factory C :  1.00

Total Shipping Cost: 240.00
```

Percentage of materials used and unit cost per customer (N)

This part needs some additional code to join var_fcp and var_sfm. However it is not overly complex. The first part is get what fraction of material is used by each factory for each customer.

```
    def get_factory_customer_material_fraction(self, fact, cust, mat):
        # Determine for each customer the fraction of each material each
        # factory has to order for manufacturing products delivered to that
        # particular customer

        def get_total_factory_material():
            total = 0.0
            for supp in self.supplier_names:
                value = self.var_sfm[supp][fact][mat].SolutionValue()
                if value > 0.0 and value != float('inf') \
                    and value != float('nan'):
                    total += float(value)
            return total

        def get_product_material_requirements(prod):
            value = get_element(self.product_requirements_df, prod, mat)
            value = float(value)
            if value == float('inf') or value == float('nan'):
                value = 0.0
            return value

        def get_total_customer_material():
            total = 0.0
            for prod in self.product_names:
                req_pu = get_product_material_requirements(prod)
                req_pu = float(req_pu)
                if req_pu == float('inf') or req_pu == float('nan'):
                    req_pu = 0.0
                qty = self.var_fcp[fact][cust][prod].SolutionValue()
                qty = float(qty)
                if qty == float('inf') or qty == float('nan'):
                    qty = 0.0
                total += (qty * req_pu)
            return total
```

```
        total_mat = get_total_factory_material()
        cust_mat = get_total_customer_material()
        if total_mat == 0.0: assert(cust_mat == 0.0)
        return cust_mat / total_mat if cust_mat > 0.0 else 0.0
```

The next part uses this information to find the average cost per customer per product.

```
    def print_factory_customer_material_fraction(self):
        print()
        print("Printing what fraction of material is used for each customer")
        print('*' * \
            len("Printing what fraction of material is used for each customer"))
        print()
        for fact in self.factory_names:
            print(f'{fact}')
            print('-' * len(f'{fact}'))
            for cust in self.customer_names:
                outstr = "    -- "
                outstr = outstr + f"{cust} :    "
                for mat in self.material_names:
                    frac = self.get_factory_customer_material_fraction(\
                                fact, cust, mat)
                    outstr = outstr + f"{mat} : {frac:5.2f}    "
                print(outstr)
```

The production cost per customer is arrived by this function. The material used by every factory for all customers is stored in a 2-D map from (factory, material) to quantity. This is used to see the fraction of each material used for each customer and that is aggregated to find the average production cost for that customer.

```
    def print_unit_product_cost_per_customer(self):

        print()
        print("Printing product unit cost per customer")
        print('*' * len("Printing product unit cost per customer"))
        print()

        all_costs = 0.0
        all_qty = 0.0

        for cust in self.customer_names:
            # 2D map for materials used
            # mat_used[factory][material] = <qty used for this customer>
            print(cust)
            print('-' * len(cust))

            for prod in self.product_names:
                qty_acc = 0.0
                cost_acc = 0.0

                for fact in self.factory_names:
                    qty = self.var_fcp[fact][cust][prod].SolutionValue()
                    if qty >= EPSILON:
                        cost = self.average_product_cost_for_factory(fact, prod)
                        ship = self.shipping_cost_factory_customer(fact, cust)
                        cost_acc += (cost * qty)
                        cost_acc += (ship * qty)
                        qty_acc += qty
```

```
                avg_prod_cost = cost_acc / qty_acc if qty_acc > 0.0 else 0.0
                if qty_acc == 0.0: assert(cost_acc == 0.0)
                if qty_acc != 0.0:
                    print(f"        {prod} : AVG COST: {avg_prod_cost:8.2f}")

                all_costs += cost_acc
                all_qty += qty_acc

        print()
        print("Total_cost ", all_costs, all_qty)
```

Output

```
Printing product unit cost per customer
***************************************

Customer A
----------
        Product A : AVG COST:   910.98
        Product D : AVG COST:  3913.75
Customer B
----------
        Product A : AVG COST:   745.71
Customer C
----------
        Product B : AVG COST:  1026.84
        Product D : AVG COST:  4003.75
Customer D
----------
        Product C : AVG COST:  2921.58
        Product D : AVG COST:  2759.01

Total_cost  49315.0 24.0

Printing what fraction of material is used for each customer
************************************************************

Factory A
---------
    -- Customer A :    Material A :  0.64    Material B :  0.62    Material C :
0.00    Material D :  0.00
    -- Customer B :    Material A :  0.13    Material B :  0.13    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.00    Material B :  0.00    Material C :
0.14    Material D :  0.10
    -- Customer D :    Material A :  0.23    Material B :  0.25    Material C :
0.86    Material D :  0.90
Factory B
---------
    -- Customer A :    Material A :  0.00    Material B :  0.00    Material C :
0.00    Material D :  0.00
    -- Customer B :    Material A :  1.00    Material B :  0.18    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.00    Material B :  0.00    Material C :
0.05    Material D :  1.00
    -- Customer D :    Material A :  0.00    Material B :  0.82    Material C :
0.95    Material D :  0.00
Factory C
```

```
---------
    -- Customer A :    Material A :  0.52    Material B :  0.50    Material C :
0.20    Material D :  0.20
    -- Customer B :    Material A :  0.00    Material B :  0.00    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.36    Material B :  0.37    Material C :
0.60    Material D :  0.60
    -- Customer D :    Material A :  0.12    Material B :  0.12    Material C :
0.20    Material D :  0.20
```

Results

The best total cost is 49315.00.

```
Printing Supplier Factory Orders
*********************************

Factory A
---------
    Supplier A -     Material A: 20.00          Material B: 20.00
    Supplier B -     Material A: 19.00          Material B: 04.00
    Supplier C -
    Supplier D -     Material C: 14.00          Material D: 50.00
    Supplier E -

Factory B
---------
    Supplier A -
    Supplier B -     Material A: 06.00          Material B: 34.00
    Supplier C -     Material C: 32.00          Material D: 05.00
    Supplier D -     Material C: 06.00
    Supplier E -     Material A: 04.00

Factory C
---------
    Supplier A -
    Supplier B -     Material B: 06.00
    Supplier C -     Material B: 10.00          Material C: 20.00
Material D: 35.00
    Supplier D -
    Supplier E -     Material A: 25.00          Material D: 40.00


Printing supplier bill for factories
************************************

Factory A
---------
    - Supplier A           :    2800.00
    - Supplier B           :    2155.00
    - Supplier C           :       0.00
    - Supplier D           :    6440.00
    - Supplier E           :       0.00

Factory B
---------
    - Supplier A           :       0.00
    - Supplier B           :    3500.00
    - Supplier C           :    8550.00
    - Supplier D           :    1380.00
    - Supplier E           :     260.00
```

```
Factory C
---------
    - Supplier A           :       0.00
    - Supplier B           :    1590.00
    - Supplier C           :   12450.00
    - Supplier D           :       0.00
    - Supplier E           :    7325.00


Printing production and cost for each factory
*********************************************

Factory A
---------
        Product A :            6.00
        Product B :            1.00
        Product C :            0.00
        Product D :            3.00
    Total Manufacturing Cost = 1010.00

Factory B
---------
        Product A :            2.00
        Product B :            1.00
        Product C :            4.00
        Product D :            0.00
    Total Manufacturing Cost = 430.00

Factory C
---------
        Product A :            2.00
        Product B :            0.00
        Product C :            0.00
        Product D :            5.00
    Total Manufacturing Cost = 425.00


Printing shipments for each customer
************************************

Customer A
----------
Product Product A          Factory A :  5.00       Factory C :  2.00
Product Product B
Product Product C
Product Product D          Factory C :  1.00

Total Shipping Cost: 280.00

Customer B
----------
Product Product A          Factory A :  1.00       Factory B :  2.00
Product Product B
Product Product C
Product Product D

Total Shipping Cost: 110.00

Customer C
----------
Product Product A
Product Product B          Factory A :  1.00       Factory B :  1.00
```

```
Product Product C
Product Product D           Factory C :  3.00

Total Shipping Cost: 370.00

Customer D
----------
Product Product A
Product Product B
Product Product C           Factory B :  4.00
Product Product D           Factory A :  3.00        Factory C :  1.00

Total Shipping Cost: 240.00



Printing product unit cost per customer
**************************************

Customer A
----------
        Product A : AVG COST:   910.98
        Product D : AVG COST:  3913.75
Customer B
----------
        Product A : AVG COST:   745.71
Customer C
----------
        Product B : AVG COST:  1026.84
        Product D : AVG COST:  4003.75
Customer D
----------
        Product C : AVG COST:  2921.58
        Product D : AVG COST:  2759.01

Total_cost  49315.0 24.0

Printing what fraction of material is used for each customer
************************************************************

Factory A
---------
    -- Customer A :    Material A :  0.64    Material B :  0.62    Material C :
0.00    Material D :  0.00
    -- Customer B :    Material A :  0.13    Material B :  0.13    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.00    Material B :  0.00    Material C :
0.14    Material D :  0.10
    -- Customer D :    Material A :  0.23    Material B :  0.25    Material C :
0.86    Material D :  0.90
Factory B
---------
    -- Customer A :    Material A :  0.00    Material B :  0.00    Material C :
0.00    Material D :  0.00
    -- Customer B :    Material A :  1.00    Material B :  0.18    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.00    Material B :  0.00    Material C :
0.05    Material D :  1.00
    -- Customer D :    Material A :  0.00    Material B :  0.82    Material C :
0.95    Material D :  0.00
Factory C
---------
```

```
    -- Customer A :    Material A :  0.52    Material B :  0.50    Material C :
0.20    Material D :  0.20
    -- Customer B :    Material A :  0.00    Material B :  0.00    Material C :
0.00    Material D :  0.00
    -- Customer C :    Material A :  0.36    Material B :  0.37    Material C :
0.60    Material D :  0.60
    -- Customer D :    Material A :  0.12    Material B :  0.12    Material C :
0.20    Material D :  0.20
```

Task 2: Modified Traveling Salesman

Initially, a full version of TSP was coded. However, the problem states that only some cities need to be visited mandatorily, and the other cities are optional. The code was then modified so that it could solve either the full TSP problem, or the reduced TSP problem.

This behaviour is controlled by the global setting FULL_TSP. By default FULL_TSP is False.

Decision Variables (A)

A 2D matrix of decision variables is created. If the salesman goes directly from A to B, then matrix[A][B] = 1, else it is 0.

```
    def create_edges(self):
        outer = {}
        for c1 in self.city_names:
            inner = {}
            for c2 in self.city_names:
                varname = f"{c1:>10s} ---> {c2:10s}"
                inner[c2] = self.solver.IntVar(0, 1, varname)
            outer[c1] = inner
        return outer
```

To ensure that there are no smaller loops, a different set of variables is created, one for each town. These variables indicate the order in which the cities are visited. A condition over these variables ensures that there are no smaller loops.

```
    # We use additional variables, each having values from 1 to N
    # if city[k] is the i'th city to be visited, then city[k] = i
    # This will help us to avoid loops
    def create_order_variables(self):
        return [self.solver.IntVar(1, len(self.city_names), f"{n}-{c}") \
            for n, c in enumerate(self.city_names)]
```

All towns must be visited maximum once, any city entered must also be left (C)

All towns must be visited no more than once, and any city entered must also be exited (except the first city). For the first city, we start with it and also end in it. Since we measure

the number of incoming and outgoing edges taken for each city, both these conditions collapse into one.

```
    def create_constraint_all_towns_visited_max_once(self):
        for c1 in self.city_names:
            cons1 = self.solver.Constraint(0, 1)
            cons2 = self.solver.Constraint(0, 1)
            for c2 in self.city_names:
                var1 = self.var_edges[c1][c2]
                cons1.SetCoefficient(var1, 1)
                var2 = self.var_edges[c2][c1]
                cons2.SetCoefficient(var2, 1)

        # A town entered must also be exited
        for c1 in self.city_names:
            cons = self.solver.Constraint(0, 0)
            for c2 in self.city_names:
                var1 = self.var_edges[c1][c2]
                cons.SetCoefficient(var1, -1)
                var2 = self.var_edges[c2][c1]
                cons.SetCoefficient(var2, 1)
```

All towns that need to be visited must be visited (B)

All towns that need to be visited must also be visited.

```
    def create_constraint_must_visit_cities(self):
        # Every city in the must-visit list must be visited
        for city in self.cities_must_visit:
            cons1 = self.solver.Constraint(1, self.solver.infinity())
            cons2 = self.solver.Constraint(1, self.solver.infinity())
            for c2 in self.city_names:
                if c2 != city:
                    var1 = self.var_edges[city][c2]
                    cons1.SetCoefficient(var1, 1)
                    var2 = self.var_edges[c2][city]
                    cons2.SetCoefficient(var2, 1)
```

Here:

```
        self.cities_must_visit = \
            ["Dublin", "Limerick", "Waterford", "Galway", "Wexford", \
            "Belfast", "Athlone", "Rosslare", "Wicklow"]
```

No disconnected self-contained cycles (D)

The order variables are used here to ensure that there are no disconnected cycles. These variables are in self.var_order, but for simplicity of notation we will name them u_0, u_1, …. The variables for each pair of town whether the leg is taken or not are denoted below as $x_{i,j}$ for ease of notation. This indicates that the edge from city I to city J is taken.

We have $u_{start} = 1$.

The condition to avoid disconnected self-contained cycles is given by the following.

$u_i - u_j + 1 <= (N - 1)(1 - x_{i,j})$ where i, j != start

This can be simplified to the following:

$2 - N <= -u_i + u_j + (1 - N)\, x_{i,j} <= \text{infinity}$, where i, j != start

This is implemented as follows:

```
    def create_constraint_no_complete_subroutes(self):
        # There should be no complete sub-routes.
        # This will be achieved by self.var_order
        # Each city is given a number in the order in which it is visited
        # There cannot be an edge from a city later in the route to a
        # city earlier in the route, except for the first and last city

        # order of first city is 1
        constraint = self.solver.Constraint(1, 1)
        constraint.SetCoefficient(self.var_order[self.start_city_ind], 1)

        for i in range(self.num_cities):
            for j in range(self.num_cities):
                if i != self.start_city_ind and j != self.start_city_ind:
                    cons = self.solver.Constraint(\
                        2 - self.num_cities, self.solver.infinity())
                    c1_name = self.city_names[i]
                    c2_name = self.city_names[j]
                    var_edge = self.var_edges[c1_name][c2_name]
                    cons.SetCoefficient(var_edge, 1 - self.num_cities)
                    cons.SetCoefficient(self.var_order[i], -1)
                    cons.SetCoefficient(self.var_order[j], 1)
```

Objective function (E)

The objective is to minimize the total distance. This is easily done by adding up all edges taken multiplied by the distance of that leg.

```
    def set_objective_coefficients(self):
        for c1 in self.city_names:
            for c2 in self.city_names:
                var = self.var_edges[c1][c2]
                dist = get_element(self.distances_df, c1, c2)
                self.objective.SetCoefficient(var, float(dist))
```

The objective is set to minimization.
`self.objective.SetMinimization()`

Solving and printing the optimal route

Solving is easily done by calling Solve(). The route is printed iteratively:

```
    def print_route(self):
        next_city = None
        current_city = self.start_city_name

        n = 0

        while self.start_city_name != next_city:
            n += 1
            if next_city != None:
                current_city = next_city
            next_city, dist = self.get_next_city(current_city)
            print(f"{n:>3d}. {current_city:>10s} ---> "\
                + f"{next_city:10s} -- {dist:>10d}")
```

Results

```
  1.       Cork ---> Waterford  --        126
  2.  Waterford ---> Rosslare   --         82
  3.   Rosslare ---> Wexford    --         19
  4.    Wexford ---> Wicklow    --         90
  5.    Wicklow ---> Dublin     --         51
  6.     Dublin ---> Belfast    --        167
  7.    Belfast ---> Athlone    --        227
  8.    Athlone ---> Galway     --         93
  9.     Galway ---> Limerick   --        105
 10.   Limerick ---> Cork       --        105

Optimal distance: 1065.0
```

The optimal distance is 1065.0.

Task 3: Planning trains

In this problem, there are three classes.

1. TrainBase: The base class for the other classes and has some common functionality and utility functions, like finding the stations in a route and so on.
2. ShortestPath: Derives from TrainBase. It is used to calculate the shortest path in minutes from one station to another.
3. TrainCapacity: Derives from TrainBase. It is used to plan the capacity. It uses ShortestPath to do its job.

Loading the input file (A)

Pandas is used to load the input file.

```
    def read_csv(self, sheet_name:str)->pd.DataFrame:
        df = pd.read_excel(self.excel_file_name, sheet_name=sheet_name)
        return df
```

Shortest paths between two stations (B)

Shortest path: decision variables (B-b)

The decision variables here are a 2-dimensional array of IntVars, which are either 0 or 1. If an edge from A to B is taken, then edge[A][B] = 1, otherwise 0.

The function also sets edge[A][B] for any two stations which are not adjacent to each other in any train line.

```
    def create_edges(self):
        # Decision variables if an edge from x to y is taken
        #    Two additional housekeeping things are done using constraints here:
        #    - An edge from a node to itself is always zero
        #    - If an edge between two nodes doesn't exist, it is always zero
        #
        outer = {}
        for st1 in self.stop_names:
            inner = {}
            for st2 in self.stop_names:
                name = f"edgetaken({st1},{st2})"
                inner[st2] = self.solver.IntVar(0, 1, name)
            outer[st1] = inner

        # Create a constraint that an edge cannot be taken from a stop
        # to itself
        for st in self.stop_names:
            var = outer[st][st]
            constraint = self.solver.Constraint(0, 0)
            constraint.SetCoefficient(var, 1)

        for st1 in self.stop_names:
            for st2 in self.stop_names:
                if float('nan') == get_element(self.distance_df, st1, st2):
                    var = outer[st1][st2]
                    constraint = self.solver.Constraint(0, 0)
                    constraint.SetCoefficient(var, 1)

        return outer
```

Shortest path: constrains (B-c)

Passengers should not backtrack the same path. If the passenger travels from A to B, he cannot also travel from B to A. This would just be inefficient. This also takes care of the condition that dead-ends not on the route are avoided, because going to a dead end would mean taking the same path back. This is specified as 0 <= edge(A,B) + edge(B,A) <= 1.

```
    def cannot_retrace_path(self):
        # If we take A->B then we cannot take B->A
        # This also takes care of dead ends
        for st1 in self.stop_names:
            for st2 in self.stop_names:
                if st1 != st2:
                    constraint = self.solver.Constraint(0, 1)
                    var_front = self.edge[st1][st2]
                    var_back = self.edge[st2][st1]
                    constraint.SetCoefficient(var_front, 1)
                    constraint.SetCoefficient(var_back, 1)
```

The source and destination must be included in the route. These are set by the following function.

```
    def set_source_destination_true(self):
        """ Set the source and destination station as taken """

        # source
        # It should be connected to exactly one station
        constraint = self.solver.Constraint(1, self.solver.infinity())
        for st in self.stop_names:
            if st != self.source:
                var = self.edge[self.source][st]
                constraint.SetCoefficient(var, 1)

        # source
        # It cannot have incoming nodes
        constraint = self.solver.Constraint(0, 0)
        for st in self.stop_names:
            if st != self.source:
                var = self.edge[st][self.source]
                constraint.SetCoefficient(var, 1)

        # destination
        # It should be connected to exactly one station
        constraint = self.solver.Constraint(1, self.solver.infinity())
        for st in self.stop_names:
            if st != self.destination:
                var = self.edge[st][self.destination]
                constraint.SetCoefficient(var, 1)

        # destination
        # It cannot have outgoing nodes
        constraint = self.solver.Constraint(0, 0)
        for st in self.stop_names:
            if st != self.destination:
                var = self.edge[self.destination][st]
                constraint.SetCoefficient(var, 1)



    def add_condition_for_intermediate_stops(self):
        # For intermediate stops, add a condition that if there is an
        # outgoing edge, there must be an incoming edge

        def constrain(stopname):
            if stopname in [self.source, self.destination]:
```

```
                return

            constraint = self.solver.Constraint(0, 0)
            # Incoming into that node
            for st in self.stop_names:
                if st != stopname:
                    var = self.edge[st][stopname]
                    constraint.SetCoefficient(var, 1)
                    var = self.edge[stopname][st]
                    constraint.SetCoefficient(var, -1)

        for stopname in self.stop_names:
            constrain(stopname)
```

Shortest path: minimize objective function (B-c)

The objective function is to minimize the total time for travel. This is easily achieved by multiplying each edge(X,Y) in the edges matrix by the distance between X and Y, and then summing up all the values. As only the edges which the passenger actually uses will be 1, this will give the travel distance. Minimizing this will reduce the traveling time.

```
    def set_objective_coefficients(self):
        # Set the coefficients of the objective. For every path that is taken
        # the weight is the same as the distance taken in the excel sheet
        maxdistance = self.get_max_size()
        maxdistance *= maxdistance
        for st1 in self.stop_names:
            for st2 in self.stop_names:
                dist = get_element(self.distance_df, st1, st2)
                if math.isnan(dist):
                    dist = maxdistance
                var = self.edge[st1][st2]
                self.objective.SetCoefficient(var, int(dist))
```

Shortest path: solving and printing (B-d)

Once the coefficients are set, solving this is easy and the solver does the work for us. The distance can be found out by querying the objective. The legs on the route and the lines connecting them are found out by these set of functions, and are relatively straight forward.

```
    def get_shortest_path(self):
        # Returns distance, [nodestart, node1, node2, ..., nodeend]
        self.solve()
        shortest_path = self.get_path(self.source, self.destination)
        stations_in_leg = {}
        for x, y in pair_array(shortest_path):
            stations_in_leg[(x,y,)] = self.get_line_for_leg(x, y)
        return self.objective.Value(), shortest_path, stations_in_leg


    def get_path(self, source, destination):
        # Returns [nodestart, node1, ..., nodeend]
        current_node = source
```

```
        next_node = None
        ret = []
        while next_node != destination:
            if next_node == None:
                ret.append(current_node)
            else:
                current_node = next_node
            next_node = self.get_next_node(current_node)
            ret.append(next_node)
        return ret

    @lru_cache(maxsize=512)
    def get_line_for_leg(self, stop1:str, stop2:str)->list:
        # Given two adjacent stations, it returns the list of the lines
        # lines connecting the stations
        line_arr = []
        for line in self.line_names:
            stations = self.get_line_stations(line)
            legs = pair_array(stations, self.is_line_circular(line))
            for x, y in legs:
                if (x == stop1 and y == stop2) or (x == stop2 and y == stop1):
                    line_arr.append(line)
        return line_arr


def pair_array(arr:list, is_circular=False)->list:
    # Given an array [1, 2, 3, 4]
    # Return an array of pairs [(1,2), (2, 3), (3,4)]
    ret = [(arr[i], arr[i+1],) for i in range(len(arr) - 1)]
    if is_circular and len(arr) >= 2:
        ret.append((arr[-1], arr[0],))
    return ret
```

Output

Since there are too many paths, the output is sampled and printed.

```
    def print_shortest_paths(self):
        for source, destination, dist, route, lines_in_leg \
            in self.shortest_path_samples:
            print(f"DISTANCE({source} --> {destination}) = {int(dist):>3d}" +\
                  f"           {route}")
            print(f"                                        {lines_in_leg}")
            print()
```

```
Printing a sample of shortest paths calculated between stations
---------------------------------------------------------------

DISTANCE(A --> C) =  10          ['A', 'B', 'C']
                                 {('A', 'B'): ['L4'], ('B', 'C'): ['L4']}

DISTANCE(B --> C) =   7          ['B', 'C']
                                 {('B', 'C'): ['L4']}

DISTANCE(B --> D) =  12          ['B', 'C', 'D']
                                 {('B', 'C'): ['L4'], ('C', 'D'): ['L3']}

DISTANCE(B --> O) =  23          ['B', 'C', 'F', 'N', 'O']
                                 {('B', 'C'): ['L4'], ('C', 'F'): ['L3', 'L4'], ('F', 'N'): ['L2'],
('N', 'O'): ['L2']}
```

```
DISTANCE(B --> Q) =  17          ['B', 'C', 'F', 'G', 'Q']
                                 {('B', 'C'): ['L4'], ('C', 'F'): ['L3', 'L4'], ('F', 'G'): ['L3',
'L4'], ('G', 'Q'): ['L1']}

DISTANCE(C --> J) =  13          ['C', 'F', 'K', 'J']
                                 {('C', 'F'): ['L3', 'L4'], ('F', 'K'): ['L2'], ('K', 'J'): ['L1']}

DISTANCE(E --> J) =  22          ['E', 'D', 'C', 'F', 'K', 'J']
                                 {('E', 'D'): ['L3'], ('D', 'C'): ['L3'], ('C', 'F'): ['L3', 'L4'],
('F', 'K'): ['L2'], ('K', 'J'): ['L1']}

DISTANCE(E --> M) =  27          ['E', 'D', 'C', 'F', 'N', 'M']
                                 {('E', 'D'): ['L3'], ('D', 'C'): ['L3'], ('C', 'F'): ['L3', 'L4'],
('F', 'N'): ['L2'], ('N', 'M'): ['L1']}

DISTANCE(F --> O) =  14          ['F', 'N', 'O']
                                 {('F', 'N'): ['L2'], ('N', 'O'): ['L2']}

DISTANCE(H --> A) =  18          ['H', 'G', 'F', 'C', 'B', 'A']
                                 {('H', 'G'): ['L3', 'L4'], ('G', 'F'): ['L3', 'L4'], ('F', 'C'):
['L3', 'L4'], ('C', 'B'): ['L4'], ('B', 'A'): ['L4']}

DISTANCE(I --> J) =   9          ['I', 'J']
                                 {('I', 'J'): ['L4']}

DISTANCE(I --> P) =  13          ['I', 'P']
                                 {('I', 'P'): ['L3']}

DISTANCE(J --> A) =  23          ['J', 'K', 'F', 'C', 'B', 'A']
                                 {('J', 'K'): ['L1'], ('K', 'F'): ['L2'], ('F', 'C'): ['L3', 'L4'],
('C', 'B'): ['L4'], ('B', 'A'): ['L4']}

DISTANCE(J --> I) =   9          ['J', 'I']
                                 {('J', 'I'): ['L4']}

DISTANCE(J --> N) =  18          ['J', 'K', 'G', 'Q', 'N']
                                 {('J', 'K'): ['L1'], ('K', 'G'): ['L1'], ('G', 'Q'): ['L1'], ('Q',
'N'): ['L1']}

DISTANCE(J --> O) =  22          ['J', 'K', 'H', 'O']
                                 {('J', 'K'): ['L1'], ('K', 'H'): ['L2'], ('H', 'O'): ['L2']}

DISTANCE(L --> D) =  25          ['L', 'J', 'K', 'F', 'C', 'D']
                                 {('L', 'J'): ['L4'], ('J', 'K'): ['L1'], ('K', 'F'): ['L2'], ('F',
'C'): ['L3', 'L4'], ('C', 'D'): ['L3']}

DISTANCE(M --> G) =  16          ['M', 'N', 'Q', 'G']
                                 {('M', 'N'): ['L1'], ('N', 'Q'): ['L1'], ('Q', 'G'): ['L1']}

DISTANCE(N --> F) =   9          ['N', 'F']
                                 {('N', 'F'): ['L2']}

DISTANCE(N --> K) =  12          ['N', 'Q', 'G', 'K']
                                 {('N', 'Q'): ['L1'], ('Q', 'G'): ['L1'], ('G', 'K'): ['L1']}

DISTANCE(O --> M) =  12          ['O', 'N', 'M']
                                 {('O', 'N'): ['L2'], ('N', 'M'): ['L1']}

DISTANCE(O --> Q) =   9          ['O', 'N', 'Q']
                                 {('O', 'N'): ['L2'], ('N', 'Q'): ['L1']}

DISTANCE(P --> D) =  29          ['P', 'I', 'H', 'G', 'F', 'C', 'D']
                                 {('P', 'I'): ['L3'], ('I', 'H'): ['L3', 'L4'], ('H', 'G'): ['L3',
'L4'], ('G', 'F'): ['L3', 'L4'], ('F', 'C'): ['L3', 'L4'], ('C', 'D'): ['L3']}

DISTANCE(P --> M) =  35          ['P', 'I', 'H', 'G', 'Q', 'N', 'M']
                                 {('P', 'I'): ['L3'], ('I', 'H'): ['L3', 'L4'], ('H', 'G'): ['L3',
'L4'], ('G', 'Q'): ['L1'], ('Q', 'N'): ['L1'], ('N', 'M'): ['L1']}

DISTANCE(P --> N) =  28          ['P', 'I', 'H', 'G', 'Q', 'N']
                                 {('P', 'I'): ['L3'], ('I', 'H'): ['L3', 'L4'], ('H', 'G'): ['L3',
'L4'], ('G', 'Q'): ['L1'], ('Q', 'N'): ['L1']}

DISTANCE(Q --> B) =  17          ['Q', 'G', 'F', 'C', 'B']
                                 {('Q', 'G'): ['L1'], ('G', 'F'): ['L3', 'L4'], ('F', 'C'): ['L3',
'L4'], ('C', 'B'): ['L4']}

DISTANCE(Q --> M) =  11          ['Q', 'N', 'M']
```

```
{('Q', 'N'): ['L1'], ('N', 'M'): ['L1']}
```

Optimal number of trains (C)

Decision Variables (C-a)

For each of the lines, the following variables are created:

1. var_upstream_trains_per_hour : list of train frequencies required per hour for each of the lines
2. var_downstream_trains_per_hour : list of trains frequencies required per hour for each of the lines downstream

The two above variables are only different for circular lines, and additional constraints are put in for lines which are not circular to make these two variables to be the same.

```
        # We will treat upstream and downstream capacity separately
        # This will give some added flexibility rather than just
        # assuming that trains upstream and downstream are the same
        #
        # This also helps in keeping the code common
        #
        # For non-circular lines, these values for upstream and downstream
        # will be the same. We will force them by our variables and constraints
        # rather than changing the numbers here
        #
        # ------------------------------------------------------------------
        # VARIABLE: frequency of trains required on each line (per hour)
        # ------------------------------------------------------------------
        self.var_upstream_trains_per_hour = {}
        self.var_downstream_trains_per_hour = {}
        for line in self.line_names:
            self.var_upstream_trains_per_hour[line] = self.solver.IntVar(\
                            0, self.solver.infinity(), f"n_trains_up({line})")
            self.var_downstream_trains_per_hour[line] = self.solver.IntVar(\
                            0, self.solver.infinity(), f"n_trains_dn({line})")
```

The above variables calculate the frequencies of the trains required to serve all passengers. However, what we really need to minimize is the total number of time. For each line, the total number of trains required is given by

n_trains = frequency * round_trip_time

Since the same train goes back and forth for non-circular lines, upstream and downstream trains should be the same. However, for circular lines, since we have trains running clockwise and anticlockwise, different number of trains can serve clockwise and anticlockwise directions.

```
        # We will treat upstream and downstream capacity separately
        # This will give some added flexibility rather than just
        # assuming that trains upstream and downstream are the same
```

```
        #
        # This also helps in keeping the code common
        #
        # For non-circular lines, these values for upstream and downstream
        # will be the same. We will force them by our variables and constraints
        # rather than changing the numbers here
        #
        # Variables, number of trains required on each line
        # For non-circular lines, upstream and downstream trains have
        # the same number as the same trains run back and forth
        # This is specified in the last condition in
        # constrain_link_trains_per_hour_and_total_trains
        #
        # These variables are simply the following equation:
        #
        # n_trains >= frequency * round_trip_time
        #
        # ---------------------------------------------------------------------
        # VARIABLE: Total number of trains on a line
        # ---------------------------------------------------------------------
        self.var_trains_on_line_up = {}
        self.var_trains_on_line_down = {}
        for line in self.line_names:
            self.var_trains_on_line_up[line] = self.solver.IntVar(\
                0, self.solver.infinity(), f"uptrains-on-line({line})")
            self.var_trains_on_line_down[line] = self.solver.IntVar(\
                0, self.solver.infinity(), f"downtrains-on-line({line})")
```

Finally, the frequencies and the number of trains are linked together by constrains, as well as the number of upstream and downstream trains for non-circular lines is set to be equal.

```
    def constrain_link_trains_per_hour_and_total_trains(self):
        for line in self.line_names:
            rtt = self.get_round_trip_time(line) / 60.0

            # var_upstream_trains_per_hour is the frequency of trains required
            # Therefore:
            # Number of trains required >= frequency * round_trip_time
            constrain = self.solver.Constraint(0, self.solver.infinity())
            up_per_hour_var = self.var_upstream_trains_per_hour[line]
            var_n_trains_up = self.var_trains_on_line_up[line]
            constrain.SetCoefficient(var_n_trains_up, 1)
            constrain.SetCoefficient(up_per_hour_var, -rtt)

            # var_downstream_trains_per_hour is the frequency of trains required
            # Therefore:
            # Number of trains required >= frequency * round_trip_time
            constrain2 = self.solver.Constraint(0, self.solver.infinity())
            down_per_hour_var = self.var_downstream_trains_per_hour[line]
            var_n_trains_down = self.var_trains_on_line_down[line]
            constrain2.SetCoefficient(var_n_trains_down, 1)
            constrain2.SetCoefficient(down_per_hour_var, -rtt)

            # For non-circular routes, the same train travels back and forth
            # hence the same number of trains upstream and downstream
            if not self.is_line_circular(line):
                constrain3 = self.solver.Constraint(0, 0)
                var = self.var_trains_on_line_up[line]
                var2 = self.var_trains_on_line_down[line]
                constrain3.SetCoefficient(var, 1)
```

```
            constrain3.SetCoefficient(var2, -1)

    def set_objective_coefficients(self):
        for line in self.line_names:
            var = self.var_trains_on_line_up[line]
            self.objective.SetCoefficient(var, 1)

            # Only add upstream and downstream to the objective for non-circular
            # routes. For circular routes, the same train runs upstream
            # and downstream, so we need to add only one
            if self.is_line_circular(line):
                var = self.var_trains_on_line_down[line]
                self.objective.SetCoefficient(var, 1)
        self.objective.SetMinimization()
```

The objective just adds up all the train requires, taking care to differentiate between circular and non-circular lines:

```
    def set_objective_coefficients(self):
        for line in self.line_names:
            var = self.var_trains_on_line_up[line]
            self.objective.SetCoefficient(var, 1)

            # Only add upstream and downstream to the objective for non-circular
            # routes. For circular routes, the same train runs upstream
            # and downstream, so we need to add only one
            if self.is_line_circular(line):
                var = self.var_trains_on_line_down[line]
                self.objective.SetCoefficient(var, 1)
        self.objective.SetMinimization()
```

Ensuring passenger demand is met (C-b)

To ensure passenger demand is met, the demand between each adjacent station must be calculated. This is done by the following steps

1. A 2-D matrix of demand is created to accumulate the demand
2. For each pair of stations, the shortest path is calculated, and the demand for each leg of that path is added to the matrix

```
    def add_requirements(self, source, destination):
        # For a source and destination, find the number of passengers
        # traveling. Calculate the shortest path, and add the same
        # number of passengers to each leg of the route
        # Basically finds the shortest path, and then adds the traffic between
        # the source and destination to each leg of the path
        if source == destination:
            return
        req = int(get_element(self.passenger_df, source, destination))
        dist, route, lines_in_leg = \
            ShortestPath(self.excel_file_name, source, destination)\
                .get_shortest_path()
        self.save_shortest_path(source, destination, dist, route, lines_in_leg)
        route = pair_array(route)
        for x, y in route:
            self.capacity_required[x][y] += req
```

Finally, constrains are added to ensure that the demand on each line is met.

```
    def add_constraints_no_leg_overloads(self):
        # Ensures that between each pair of connected stations,
        # there are enough trains to carry people to meet the demand
        #
        # The demand has already been accumulated
        # if A -> B -> C, and we are considering B -> C,
        # both B -> C and A -> C have already been added
        #
        # Also A -> A = 0, we could have added an if condition to remove this
        # constraint, but its just easier to let it be
        def constrain(source, dest):
            # Requirement(A to B) <= sum_over_lines(line capacity from A to B)
            req = self.capacity_required[source][dest]
            cons = self.solver.Constraint(int(req), self.solver.infinity())
            for line in self.line_names:
                capacity = self.line_capacity_per_train_up[line][source][dest]
                var = self.var_upstream_trains_per_hour[line]
                cons.SetCoefficient(var, int(capacity))

                capacity = self.line_capacity_per_train_down[line][source][dest]
                var = self.var_downstream_trains_per_hour[line]
                cons.SetCoefficient(var, int(capacity))

        for s1 in self.stop_names:
            for s2 in self.stop_names:
                constrain(s1, s2)
```

The above takes care that if multiple lines are available, then the passengers will use both the lines since we sum over all lines. In the above, if A and B are not adjacent stations, the demand in line_capacity_per_train_up and line_capacity_per_train_down will be 0. As such many of the varialbes are redundant, but it was just easier to write code with redundant variables. They will not have an effect anyway.

Objective function (C-c)

The objective just adds up all the train requires, taking care to differentiate between circular and non-circular lines:

```
    def set_objective_coefficients(self):
        for line in self.line_names:
            var = self.var_trains_on_line_up[line]
            self.objective.SetCoefficient(var, 1)

            # Only add upstream and downstream to the objective for non-circular
            # routes. For circular routes, the same train runs upstream
            # and downstream, so we need to add only one
            if self.is_line_circular(line):
                var = self.var_trains_on_line_down[line]
                self.objective.SetCoefficient(var, 1)
        self.objective.SetMinimization()
```

Additional functions

The capacity of each line is between adjacent stations is calculated and stored:

```
    def fill_line_capacity_per_train(self, line:str):
        # This matrix denotes the capacity for each line, that is
        # how many passengers can a single train carry from A to B in line L
        # where A and B are adjacent stations in line L
        #
        # For non-circular lines, one of upstream or downstream from A to B
        # will be 0
        capacity = get_element(self.train_capacity_df, line, 'Capacity')
        if self.is_line_circular(line):
            for x, y in self.station_pairs(line):
                self.line_capacity_per_train_up[line][x][y] = capacity
            for x, y in self.station_pairs(line, downstream=True):
                self.line_capacity_per_train_down[line][x][y] = capacity
        else:
            upstream_stations = self.get_line_stations(line)
            upstream_stations = pair_array(upstream_stations)
            downstream_stations = self.get_line_stations(line)[::-1]
            downstream_stations = pair_array(downstream_stations)
            for x, y in upstream_stations:
                self.line_capacity_per_train_up[line][x][y] = capacity
            for x, y in downstream_stations:
                self.line_capacity_per_train_down[line][x][y] = capacity
```

All requirements in terms of train frequency for each pair of adjacent station is added here:

```
    def add_requirements(self, source, destination):
        # For a source and destination, find the number of passengers
        # traveling. Calculate the shortest path, and add the same
        # number of passengers to each leg of the route
        # Basically finds the shortest path, and then adds the traffic between
        # the source and destination to each leg of the path
        if source == destination:
            return
        req = int(get_element(self.passenger_df, source, destination))
        dist, route, lines_in_leg = \
            ShortestPath(self.excel_file_name, source, destination)\
                .get_shortest_path()
        self.save_shortest_path(source, destination, dist, route, lines_in_leg)
        route = pair_array(route)
        for x, y in route:
            self.capacity_required[x][y] += req

    def add_all_requirements(self):
        # Add the requirements for each pair of adjacent stations
        # Basically finds the shortest path, and then adds the traffic between
        # the source and destination to each leg of the path
        description = "Finding shortest paths"
        pairs = [(s1, s2,) for s1 in self.stop_names for s2 in self.stop_names]
        for s1, s2 in tqdm(pairs, desc=description):
            self.add_requirements(s1, s2)
```

Printing the solution (C-d)

Printing the required output is quite easy now as we have everything calculated

```
    def solve(self):
        if not self.solver_invoked:
            self.solver.Solve()
        self.solver_invoked = True

    def print_solution(self):
        self.solve()

        for line in self.line_names:
            uptrains = self.var_trains_on_line_up[line].SolutionValue()
            downtrains = self.var_trains_on_line_down[line].SolutionValue()
            if self.is_line_circular(line):
                print(f"{line}:    UP: {uptrains}    DOWN: {downtrains}")
            else:
                print(f"{line}:    UP: {uptrains}")
        print()
        print(f"Total number of trains required: {self.objective.Value()}")
        print()

    def main(self):
        self.solve()
        print()
        print("Printing a sample of shortest paths calculated between stations")
        print("---------------------------------------------------------------")
        print()
        self.print_shortest_paths()
        print()
        print("Printing number of trains needed")
        print("--------------------------------")
        print()
        self.print_solution()
        print()
```

Results

```
Printing number of trains needed
--------------------------------

L1:    UP: 17.0
L2:    UP: 13.0    DOWN: 10.0
L3:    UP: 29.0
L4:    UP: 18.0

Total number of trains required: 87.0
```