Decision Analytics: Assignment 2

Linear Programming

Rajbir Bhattacharjee

R00195734

Table of Contents

[Task 1: Production Planning 2](#_Toc90595459)

[Loading the data (A) 2](#_Toc90595460)

[Decision variables (B) 3](#_Toc90595461)

[Constraints with suppliers and factories (E, F) 4](#_Toc90595462)

[Constraints with Factories and Customers (C, D, G) 5](#_Toc90595463)

[Setting up the objective function (H, I) 6](#_Toc90595464)

[Determining how much material has to be ordered from each supplier (J) 8](#_Toc90595465)

[Output 8](#_Toc90595466)

[Printing Supplier Bill for Each Factory (K) 9](#_Toc90595467)

[Output 9](#_Toc90595468)

[Units produced per factory and manufacturing cost (L) 10](#_Toc90595469)

[Output 10](#_Toc90595470)

[Number of units shipped from each factory to each customer, and shipping costs (M) 11](#_Toc90595471)

[Output 11](#_Toc90595472)

[Percentage of materials used and unit cost per customer (N) 12](#_Toc90595473)

[Output 14](#_Toc90595474)

[Results 15](#_Toc90595475)

[Task 2: Modified Traveling Salesman 18](#_Toc90595476)

[Decision Variables (A) 18](#_Toc90595477)

[All towns must be visited maximum once, any city entered must also be left (C) 18](#_Toc90595478)

[All towns that need to be visited must be visited (B) 19](#_Toc90595479)

[No disconnected self-contained cycles (D) 19](#_Toc90595480)

[Objective function (E) 20](#_Toc90595481)

[Solving and printing the optimal route 21](#_Toc90595482)

[Results 21](#_Toc90595483)

[Task 3: Planning trains 21](#_Toc90595484)

[Loading the input file (A) 22](#_Toc90595485)

[Shortest paths between two stations (B) 22](#_Toc90595486)

[Shortest path: decision variables (B-b) 22](#_Toc90595487)

[Shortest path: constrains (B-c) 22](#_Toc90595488)

[Shortest path: minimize objective function (B-c) 24](#_Toc90595489)

[Shortest path: solving and printing (B-d) 24](#_Toc90595490)

[Output 25](#_Toc90595491)

[Optimal number of trains (C) 27](#_Toc90595492)

[Decision Variables (C-a) 27](#_Toc90595493)

[Ensuring passenger demand is met (C-b) 29](#_Toc90595494)

[Objective function (C-c) 30](#_Toc90595495)

[Additional functions 31](#_Toc90595496)

[Printing the solution (C-d) 32](#_Toc90595497)

[Results 32](#_Toc90595498)

# 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 u0, u1, …. The variables for each pair of town whether the leg is taken or not are denoted below as xi,j for ease of notation. This indicates that the edge from city I to city J is taken.

We have ustart = 1.

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

ui – uj + 1 <= (N – 1)(1 – xi,j) where i, j != start

This can be simplified to the following:

2 – N <= -ui + uj +(1 – N) xi,j <= 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