Decision Analytics: Assignment 2

Linear Programming

Table of Contents

[Task 1: Production Planning 2](#_Toc90578406)

[Loading the data (A) 2](#_Toc90578407)

[Decision variables (B) 2](#_Toc90578408)

[Constraints with suppliers and factories (E, F) 3](#_Toc90578409)

[Constraints with Factories and Customers (C, D, G) 4](#_Toc90578410)

[Setting up the objective function (H, I) 5](#_Toc90578411)

[Determining how much material has to be ordered from each supplier (J) 7](#_Toc90578412)

[Output 7](#_Toc90578413)

[Printing Supplier Bill for Each Factory (K) 8](#_Toc90578414)

[Output 8](#_Toc90578415)

[Units produced per factory and manufacturing cost (L) 9](#_Toc90578416)

[Output 9](#_Toc90578417)

[Number of units shipped from each factory to each customer, and shipping costs (M) 10](#_Toc90578418)

[Output 10](#_Toc90578419)

[Percentage of materials used and unit cost per customer (N) 11](#_Toc90578420)

[Output 13](#_Toc90578421)

[Results 14](#_Toc90578422)

[Task 2: Modified Traveling Salesman 17](#_Toc90578423)

[Decision Variables (A) 17](#_Toc90578424)

[All towns must be visited maximum once, any city entered must also be left (C) 18](#_Toc90578425)

[All towns that need to be visited must be visited (B) 18](#_Toc90578426)

[No disconnected self-contained cycles (D) 19](#_Toc90578427)

[Objective function (E) 19](#_Toc90578428)

[Solving and printing the optimal route 20](#_Toc90578429)

[Output 20](#_Toc90578430)

[Task 3: Planning trains 20](#_Toc90578431)

# 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}")

## Output

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