COMP 9057 - DECISION ANALYTICS

**ASSIGNMENT 2 - Linear Programming (Report)**

# TASK 1 (supply chain, 21 points)

## Solution

* Load the spreadsheet and import the data.

I have used openpyxl library in python and defined load\_sheet function (from line 6 to line 20) to load the sheets to supplier\_stock, raw\_mat\_costs, raw\_mat\_ship, prod\_req, prod\_cap, prod\_cost, customer\_demand and ship\_cost variables (from line 53 to 68).

* And I created decision variables for the orders from the suppliers, production volume and delivery to customers as supplier\_order\_vars, production\_vol\_vars, customer\_delivery\_vars.

    # 2. Define the variables

    # supplier order variables

    supplier\_order\_vars = {}

    for supplier in suppliers:

        for material in materials:

            for factory in factories:

                supplier\_order\_vars[supplier, material, factory] = solver.IntVar(0, solver.infinity(), f'supplier\_order[{supplier}, {material}, {factory}]')

    # production volume variables

    production\_vol\_vars = {}

    for product in products:

        for factory in factories:

            production\_vol\_vars[product, factory] = solver.IntVar(0, solver.infinity(), f'production\_volume[{product}, {factory}]')

    # customer delivery variables

    customer\_delivery\_vars = {}

    for customer in customers:

        for product in products:

            for factory in factories:

                customer\_delivery\_vars[customer, product, factory] =\

                    solver.IntVar(0, solver.infinity(), f'customer\_delivery[{customer}, {product}, {factory}]')

* Then, I implemented each constraint using solver.Add function.
* Ensure factories produce more than they ship to the customers

For all products and factories, the production volume var must be greater or equal than the sum of customer delivery amount for all customers.

for product in products:

        for factory in factories:

            solver.Add(production\_vol\_vars[product, factory] - sum([customer\_delivery\_vars[customer, product, factory] for customer in customers]) >= 0)

* Ensure that customer demand is met

For all customer and product, the sum of customer delivery amount for all factories must greater or equal than the customer demand.

for customer in customers:

    for product in products:

        solver.Add(sum([customer\_delivery\_vars[customer, product, factory] for factory in factories]) >= customer\_demand[product, customer])

* Ensure that suppliers have all ordered items in stock

 # 5. Define and implement the constraints that ensure that suppliers have all ordered items in stock

    for supplier in suppliers:

        for material in materials:

            solver.Add(sum([supplier\_order\_vars[supplier, material, factory] for factory in factories]) <= supplier\_stock[supplier, material])

for material in materials:

        for factory in factories:

            solver.Add(sum([supplier\_order\_vars[supplier, material, factory] for supplier in suppliers]) -\

                       sum([production\_vol\_vars[product, factory] \* prod\_req[product, material] for product in products]) >= 0)

* Ensure that the manufacturing capacities are not exceeded

For all products and factories, the production volume must be less or equal than product capacity.

 # 6. Define and implement the constraints that ensure that the manufacturing capacities are not exceeded

    for product in products:

        for factory in factories:

            solver.Add(production\_vol\_vars[product, factory] <= prod\_cap[product, factory])

* Then, finally I have defined the objective function.

overall\_cost = supplier\_cost + supplier\_ship\_cost + production\_cost + customer\_ship\_cost

solver.Minimize(overall\_cost)

Here, I defined supplier\_cost, supplier\_ship\_cost, production\_cost, customer\_ship\_cost separately(from line 124 to line 135).

* Then solved the program using solver.Solve (line 141).

## Result

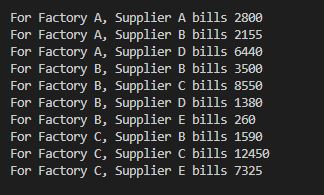
I printed each required results on the optimal solution.

The overall cost was: 49315

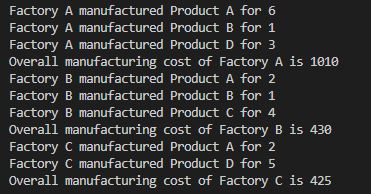
* Determine for each factory how much material has to be ordered from each individual supplier.



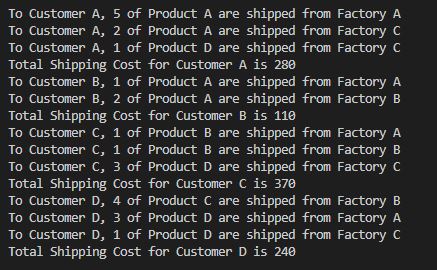
* Determine for each factory what the supplier bill comprising material cost and delivery will be for each supplier.



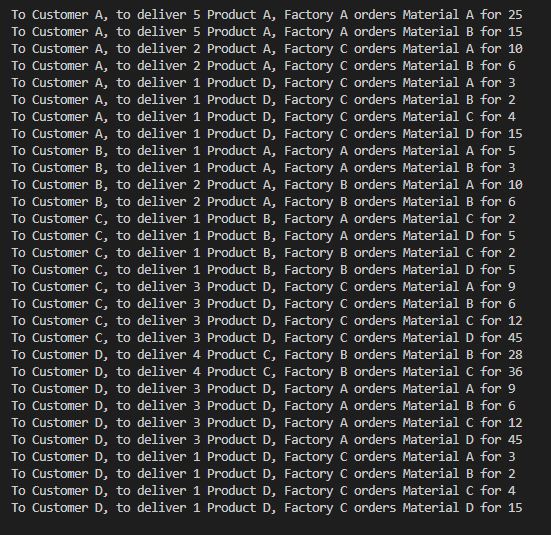
* Determine for each factory how many units of each product are being manufactured. Also determine the total manufacturing cost for each individual factory.



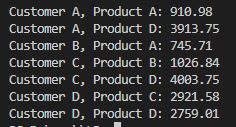
* Determine for each customer how many units of each product are being shipped from each factory. Also determine the total shipping cost per customer.



* Determine for each customer the fraction of each material each factory has to order for manufacturing products delivered to that particular customer.



* Based on this calculate the overall unit cost of each product per customer including the raw materials used for the manufacturing of the customer’s specific product, the cost of manufacturing for the specific customer and all relevant shipping costs.



# TASK 2 (delivery driver, 8 points)

## Solution

I used OR Tools’ CBC\_MIXED\_INTEGER\_PROGRAMMING solver to solve this problem.

* I loaded the xlsx spread sheet using load\_sheet function defined previously to defined distance var. (line 236 – 242)
* Defined the decision variable with 0 or 1 integer var for all pairs of towns that are not the equal.

# 2. For each pair of towns that need to be visited create a decision variable to decide if this leg should be included into the route

    legs = {}

    for town1 in towns\_to\_visit:

        for town2 in towns\_to\_visit:

            if town1 != town2:

                legs[town1, town2] = solver.IntVar(0, 1, "")

* And defined each constraint.
* Ensure that the delivery driver arrives in each of the towns that need to be visited.

For all towns, the sum of arrival leg count must be exactly 1.

# 3. Define and implement the constraints that ensure that the delivery driver arrives in each of the towns that need to be visited

        solver.Add(sum(legs[town, town2] for town2 in towns\_to\_visit if town2 != town) == 1)

* Ensure that the driver departs each of the towns that need to be visited.

The same as previous, though the sum of departure leg count must be exactly 1.

# 4. Define and implement the constraints that ensure that the driver departs each of the towns that need to be visited

        solver.Add(sum(legs[town1, town] for town1 in towns\_to\_visit if town != town1) == 1)

* Ensure that there are no disconnected self-contained circles in the route.

For all subtown collections, I defined that the count of all legs must less than the subtown count.

# 5. Define and implement the constraints that ensure that there are no disconnected self-contained circles in the route

    subtowns = [subtown for i in range(2, len(towns\_to\_visit)) for subtown in combinations(towns\_to\_visit, i)]

    for subtown in subtowns:

        solver.Add(sum(legs[town1, town2] for town1 in subtown for town2 in subtown if town1 != town2) <= len(subtown) - 1)

* And defined the objective function to be overall distance must be minimized.

# 6. Define and implement the objective function to minimise the overall distance travelled.

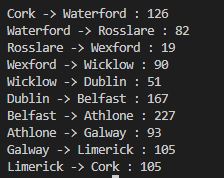
    overall\_distance = sum(legs[town1, town2] \* distance[town1, town2] for town1 in towns\_to\_visit for town2 in towns\_to\_visit if town1 != town2)

    solver.Minimize(overall\_distance)

## Result

The result was as below and the shortest distance was 1065

The path was:



# TASK 3 (investment portfolio, 21 points)

## Solution

* I load the xlsx spreadsheet using load\_sheet\_3 function (defined from line 21 to 36), then convert it USD or EUR consistently, calculate the monthly return, and overall monthly average reward for single position.
* Load the spreadsheet (from line 289 to 302). To variables usd\_data, eur\_data, currency\_data, timestamps, stocks
* Convert the currency (from line 308 to 313).

if currency == 'USD':

        for key, value in eur\_data.items():

            eur\_data[key] = currency\_data[key[0], 'EURUSD'] \* value

    else:

        for key, value in usd\_data.items():

            usd\_data[key] = value / currency\_data[key[0], 'EURUSD']

* Calculate the return data for each month (except the first month).

# calculate the monthly return

    return\_data = {}

    for i in range(1, len(timestamps)):

        for stock in stocks:

            return\_data[timestamps[i], stock] = \

                stocks\_data[timestamps[i], stock] /stocks\_data[timestamps[i - 1], stock]

    print(f"task3\_A\_{currency}")

* And calculate the overall monthly average reward for each single position.

for stock in stocks:

        average\_reward\_data[stock] = sum([return\_data[timestamp, stock] for timestamp in timestamps[1:]]) / (len(timestamps) - 1)

        print(f"The overall average monthly reward of {stock} is {average\_reward\_data[stock]}")

* I dig into Solution B part to determine the optimal marketing time. I defined the GLOP solver.
* Defined the decision variable that indicates the percentage between 0 to 1 for all positions. (positions would be stocks + ‘Cash’).

# For each month create decision variables that indicate the percentage of each position held as well as

    # the percentage of cash not invested during this month

    percent\_var = {}

    positions = stocks + ['Cash']

    for timestamp in timestamps:

        for position in positions:

            percent\_var[timestamp, position] = solver1.NumVar(0, 1, "")

* And defined the constraint to ensure that the investment portfolio always adds up to 100%.

# Identify and create the implicit constraints to ensure that the investment portfolio always adds up to 100%

    for timestamp in timestamps:

        solver1.Add(sum(percent\_var[timestamp, position] for position in positions) == 1.0)

* Defined the constraint to make each position not exceed 30% so that in this case would be 0.3.

So for each timestamp and position, the percentage var must be equal or less than 0.3.

# Investing everything into one single position is not good practice.

# Therefore, identify and create constraints that ensure that no single investment position is ever more than 30% of the overall portfolio

    for timestamp in timestamps:

        for position in positions:

            solver1.Add(percent\_var[timestamp, position] <= 0.3)

* Finally, I defined the objective function to maximize the summing up all the monthly returns.

For that, I first for each timestamp, implemented weighed sum with the percentage var for the return values calculated for each stock. (For cash, there would be no return)

return\_vars = {}

    for timestamp in timestamps[1:]:

        return\_vars[timestamp] = sum(percent\_var[timestamp, stock] \* return\_data[timestamp, stock] for stock in stocks)

Then, define the objective function to maximize the sum of return variables for all timestamps.

solver1.Maximize(sum(return\_vars[timestamp] for timestamp in timestamps[1:]))

* I solved the C part using GLOP solver to minimize the risk.
* Created the decision variable for percentage of each stock(no cash) on portfolio\_vars which are Numerical Variables between 0 to 1.

# Create decision variables that indicate the percentage of each position held in the portfolio during the entire investment period

    portfolio\_vars = {}

    for timestamp in timestamps:

        for stock in stocks:

            portfolio\_vars[timestamp, stock] = solver2.NumVar(0, 1, "")

* Then, defined the constraint that the investment portfolio always adds up to 100%.

For all timestamps, sum of portfolio vars of all stocks must be exactly 1.

# Create the implicit constraint that the investment portfolio always adds up to 100%

    for timestamp in timestamps:

        solver2.Add(sum(portfolio\_vars[timestamp, stock] for stock in stocks) == 1.0)

* And similarly, defined the constraint that all position percentage must not exceed 30%.

# Identify and create constraints to ensure that no single investment position is ever more than 30% of the overall portfolio

    for timestamp in timestamps:

        for stock in stocks:

            solver2.Add(portfolio\_vars[timestamp, stock] <= 0.3)

* And the constraint that the overall average monthly reward must exceed 0.5% so that in that case this would be 1.005.

# Create a constraint to ensure that the overall average monthly reward of the portfolio is

# at least 0.5% over the five-year investment period

    solver2.Add(sum(sum(portfolio\_vars[timestamp, stock] \* return\_data[timestamp, stock] for stock in stocks) for timestamp in timestamps[1:])\

                    / (len(timestamps) - 1) >= 1.005)

* I created additional variables to determine the boundaries of risks to bounce\_vars.

# Create these additional variables

    bounce\_vars = {}

    for timestamp in timestamps[1:]:

        bounce\_vars[timestamp] = solver2.NumVar(0, solver2.infinity(), "")

* And defined the bounding deviation constraints.

# implement the necessary constraints for bounding the deviation

    for timestamp in timestamps[1:]:

        solver2.Add(sum(portfolio\_vars[timestamp, stock] \* (return\_data[timestamp, stock] - average\_reward\_data[stock]) for stock in stocks) \

                    >= -bounce\_vars[timestamp])

        solver2.Add(sum(portfolio\_vars[timestamp, stock] \* (return\_data[timestamp, stock] - average\_reward\_data[stock]) for stock in stocks) \

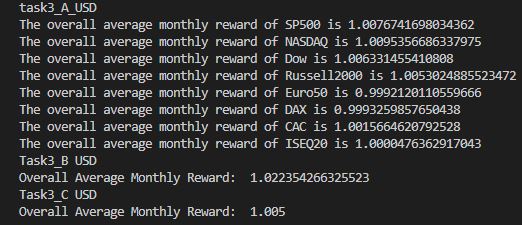
                    <= bounce\_vars[timestamp])

* Finally, minimize the boundary vars.

solver2.Minimize(sum(bounce\_vars[timestamp] for timestamp in timestamps[1:]))

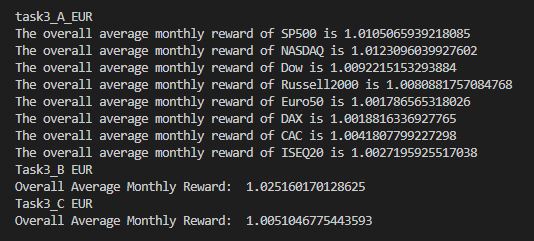
## Result

* USD result.



The position values are stored in task3\_B\_USD.csv, task3\_C\_USD.csv.

* EUR result.



The position values are stored in task3\_B\_EUR.csv, task3\_C\_EUR.csv.