OR_Assignment_3

November 3, 2025

###Question - 1(Process Flexibility)

[1]:
```python
from gurobipy import *
import numpy as np

######### Parameters Setup #########

# Supply capacity per plant
supply = np.array([100, 100, 100, 100, 100, 100])

# Initial demand mean and covariance
mean = np.array([100, 100, 100, 100, 100, 100])
cov = np.diag([900, 900, 900, 900, 900, 900])  # independent, sd = 30

Sample_Size = 1000  # number of demand samples


######### Model Definition #########
def model_setup():
    m = Model("Process_Flexi")

    # decision variables only for allowed arcs
    x = m.addVars(ARCS, name="x")

    # objective: maximize total sales
    m.setObjective(quicksum(x[i, j] for (i, j) in ARCS), GRB.MAXIMIZE)

    # capacity constraints
    m.addConstrs(
        (quicksum(x[i, j] for (i, j) in ARCS.select(i, "*")) <= supply[i]
         for i in range(len(supply))),
        "capacity"
    )

    # demand constraints
    m.addConstrs(
        (quicksum(x[i, j] for (i, j) in ARCS.select("*", j)) <= demand[j]
         for j in range(len(mean))),
        "demand"
    )

    # suppress solver output
    m.setParam("OutputFlag", False)
    return m


######### Helper Function #########
def simulate_flexibility(ARCS_input, label):
    global ARCS
    ARCS = tuplelist(ARCS_input)
    sales = np.zeros(Sample_Size)

    for i in range(Sample_Size):
        # sample random demand (truncated above 0)
        demand_sample = np.maximum(np.random.multivariate_normal(mean, cov), 0)
        globals()['demand'] = demand_sample  # update global demand

        m = model_setup()
        m.optimize()
        sales[i] = m.objVal

    avg_sales = np.average(sales)
    print(f"Average maximum sales for {label} system: {avg_sales:.2f}")
    return avg_sales


######### Dedicated System #########
ARCS_ded = [(0,0), (1,1), (2,2), (3,3), (4,4), (5,5)]
avg_ded = simulate_flexibility(ARCS_ded, "Dedicated")


######### Open Chain System #########
# Each plant produces its own product + next one (no wrap-around)
ARCS_open = [
    (0,0),(0,1),
    (1,1),(1,2),
    (2,2),(2,3),
    (3,3),(3,4),
    (4,4),(4,5),
    (5,5)
]
avg_open = simulate_flexibility(ARCS_open, "Open Chain")


######### Long Chain System #########
```

```
# Same as Open Chain but last plant also produces first product (wrap-around)
ARCS_long = ARCS_open + [(5,0)]
avg_long = simulate_flexibility(ARCS_long, "Long Chain")


######### Summary #########
print("\n=== Summary of Average Maximum Sales ===")
print(f"Dedicated : {avg_ded:.2f}")
print(f"Open Chain: {avg_open:.2f}")
print(f"Long Chain: {avg_long:.2f}")
```

```
Set parameter Username
Set parameter LicenseID to value 2697524
Academic license - for non-commercial use only - expires 2026-08-21
Average maximum sales for Dedicated system: 528.91
Average maximum sales for Open Chain system: 557.93
Average maximum sales for Long Chain system: 571.24

=== Summary of Average Maximum Sales ===
Dedicated : 528.91
Open Chain: 557.93
Long Chain: 571.24
```

###Question-2

[15]:
```
import numpy as np
import time

# Try Gurobi import
try:
    import gurobipy as gp
    from gurobipy import GRB
    GUROBI_AVAILABLE = True
except ImportError:
    GUROBI_AVAILABLE = False
    print("Gurobi not available - Part (c) & (d) require Gurobi.")


# ================================================================
# GLOBAL PARAMETERS
# ================================================================
LOCATIONS = ["Jurong West", "Orchard", "Harbour Front"]
N_LOC = len(LOCATIONS)

PRICE_PER_UNIT = 100.0
COST_PER_UNIT = 50.0
FIXED_TRANSSHIP_COST = 200.0
```

```
MEAN_DEMAND = np.array([300, 500, 500])
STD_DEV = np.array([20, 20, 40])
COV_MATRIX = np.diag(STD_DEV ** 2)

INVENTORY_ABC = np.array([300, 500, 500])
TOTAL_INVENTORY_ABC = np.sum(INVENTORY_ABC)
PROCUREMENT_COST_ABC = TOTAL_INVENTORY_ABC * COST_PER_UNIT

TRANSSHIP_COSTS = np.array([
    [0, 22, 19],
    [22, 0, 7],
    [19, 7, 0]
])

# Required in assignment
SAMPLE_SIZE_ABC = 5000   # for (a),(b),(c)
SAMPLE_SIZE_D = 50       # for (d) SAA (lower to avoid crashing)


# ================================================================
# PART (a): Expected Profit (No Transshipment)
# ================================================================
def solve_part_a():
    print("\n================ PART (a): Expected Profit (No Transshipment)␣
 ↪================")

    sales_samples = np.zeros(SAMPLE_SIZE_ABC)
    sales_loc_samples = np.zeros((SAMPLE_SIZE_ABC, N_LOC))

    for k in range(SAMPLE_SIZE_ABC):
        demand = np.maximum(np.random.multivariate_normal(MEAN_DEMAND,␣
 ↪COV_MATRIX), 0)
        sales = np.minimum(demand, INVENTORY_ABC)
        sales_samples[k] = np.sum(sales)
        sales_loc_samples[k] = sales

    avg_sales_total = np.mean(sales_samples)
    avg_sales_loc = np.mean(sales_loc_samples, axis=0)

    revenue = avg_sales_total * PRICE_PER_UNIT
    profit = revenue - PROCUREMENT_COST_ABC

    print(f"Expected Total Sales: {avg_sales_total:.2f}")
    for i in range(N_LOC):
        print(f"  {LOCATIONS[i]} Sales: {avg_sales_loc[i]:.2f}")

    print(f"Expected Revenue: ${revenue:.2f}")
    print(f"Procurement Cost: ${PROCUREMENT_COST_ABC:.2f}")
    print(f"Expected Profit (a): ${profit:.2f}")

    return profit


# ================================================================
# PART (b): Deterministic Case
# ================================================================
def solve_part_b():
    print("\n================ PART (b): Deterministic Profit ================")

    sales_loc = INVENTORY_ABC.copy()
    revenue = np.sum(sales_loc) * PRICE_PER_UNIT
    profit = revenue - PROCUREMENT_COST_ABC

    for i in range(N_LOC):
        print(f"  {LOCATIONS[i]} Sales: {sales_loc[i]}")

    print(f"Revenue: ${revenue:.2f}")
    print(f"Procurement Cost: ${PROCUREMENT_COST_ABC:.2f}")
    print(f" Profit (b): ${profit:.2f}")


# ================================================================
# PART (c): With Transshipment
# ================================================================
def solve_stage2(demand, s):
    m = gp.Model()
    m.setParam('OutputFlag', 0)

    T = m.addVars(N_LOC, N_LOC, lb=0.0)
    y = m.addVars(N_LOC, lb=0.0)
    I = m.addVars(N_LOC, lb=0.0)

    m.setObjective(
        gp.quicksum(PRICE_PER_UNIT * y[i] for i in range(N_LOC)) -
        gp.quicksum(TRANSSHIP_COSTS[i, j] * T[i, j] for i in range(N_LOC) for j␣
 ↪in range(N_LOC)),
        GRB.MAXIMIZE
    )

    for i in range(N_LOC):
        m.addConstr(y[i] <= demand[i])
        inflow = gp.quicksum(T[j, i] for j in range(N_LOC))
        outflow = gp.quicksum(T[i, j] for j in range(N_LOC))
        m.addConstr(s[i] + inflow - outflow == y[i] + I[i])
    m.optimize()

    revenue = sum(PRICE_PER_UNIT * y[i].X for i in range(N_LOC))
    var_cost = sum(TRANSSHIP_COSTS[i, j] * T[i, j].X
                   for i in range(N_LOC) for j in range(N_LOC))
    sales_loc = np.array([y[i].X for i in range(N_LOC)])
    return revenue, var_cost, sales_loc


def solve_part_c(profit_a):
    if not GUROBI_AVAILABLE:
        print("\nSkipping (c)   Gurobi not installed")
        return

    print("\n================ PART (c): With Transshipment Service␣
 ↪================")

    profit_samples = np.zeros(SAMPLE_SIZE_ABC)
    sales_loc_samples = np.zeros((SAMPLE_SIZE_ABC, N_LOC))

    for k in range(SAMPLE_SIZE_ABC):
        demand = np.maximum(np.random.multivariate_normal(MEAN_DEMAND,␣
 ↪COV_MATRIX), 0)
        revenue, var_cost, sales_loc = solve_stage2(demand, INVENTORY_ABC)
        fixed_cost = FIXED_TRANSSHIP_COST if var_cost > 0.001 else 0
        profit_samples[k] = revenue - PROCUREMENT_COST_ABC - var_cost -␣
 ↪fixed_cost
        sales_loc_samples[k] = sales_loc

    avg_profit = np.mean(profit_samples)
    avg_sales_loc = np.mean(sales_loc_samples, axis=0)

    print("Average Sales per Location:")
    for i in range(N_LOC):
        print(f"  {LOCATIONS[i]}: {avg_sales_loc[i]:.2f}")

    print(f"Average Profit (c): ${avg_profit:.2f}")
    print(f"Profit change vs (a): ${avg_profit - profit_a:.2f}")
    print("Decision:", "Adopt service" if avg_profit > profit_a else " Do not␣
 ↪adopt")


# ================================================================
# PART (d): Optimal Inventory (SAA)
# ================================================================
def solve_part_d():
    if not GUROBI_AVAILABLE:
        print("\nSkipping (d) Gurobi not installed")
        return

    print("\n================ PART (d): Optimal Inventory (SAA)␣
↪================")
    print(f"Using SAMPLE_SIZE_D = {SAMPLE_SIZE_D} (reduced to avoid crash)")

    demands = np.maximum(
        np.random.multivariate_normal(MEAN_DEMAND, COV_MATRIX, SAMPLE_SIZE_D),
        0
    )

    m = gp.Model()
    m.setParam('OutputFlag', 0)

    s = m.addVars(N_LOC, lb=0.0)
    T = m.addVars(SAMPLE_SIZE_D, N_LOC, N_LOC, lb=0.0)
    y = m.addVars(SAMPLE_SIZE_D, N_LOC, lb=0.0)
    I = m.addVars(SAMPLE_SIZE_D, N_LOC, lb=0.0)

    m.setObjective(
        (1 / SAMPLE_SIZE_D) * gp.quicksum(
            PRICE_PER_UNIT * y[k, i] - gp.quicksum(TRANSSHIP_COSTS[i, j] * T[k,␣
↪i, j]
                                                   for j in range(N_LOC))
            for k in range(SAMPLE_SIZE_D) for i in range(N_LOC)
        ) -
        gp.quicksum(COST_PER_UNIT * s[i] for i in range(N_LOC)),
        GRB.MAXIMIZE
    )

    for k in range(SAMPLE_SIZE_D):
        for i in range(N_LOC):
            m.addConstr(y[k, i] <= demands[k, i])
            inflow = gp.quicksum(T[k, j, i] for j in range(N_LOC))
            outflow = gp.quicksum(T[k, i, j] for j in range(N_LOC))
            m.addConstr(s[i] + inflow - outflow == y[k, i] + I[k, i])

    m.optimize()

    print(f"Expected Profit (d): ${m.ObjVal:.2f}")
    print("Optimal Inventory per Location:")
    for i in range(N_LOC):
        print(f"  {LOCATIONS[i]}: {s[i].X:.2f}")
    print(f"Total Procured: {sum(s[i].X for i in range(N_LOC)):.2f}")
```

```python
# ================================================================
# MAIN RUN
# ================================================================
if __name__ == "__main__":
    profit_a = solve_part_a()
    solve_part_b()
    solve_part_c(profit_a)
    solve_part_d()
```

```
================ PART (a): Expected Profit (No Transshipment) ================
Expected Total Sales: 1268.43
  Jurong West Sales: 292.14
  Orchard Sales: 492.08
  Harbour Front Sales: 484.21
Expected Revenue: $126843.08
Procurement Cost: $65000.00
Expected Profit (a): $61843.08

================ PART (b): Deterministic Profit ================
  Jurong West Sales: 300
  Orchard Sales: 500
  Harbour Front Sales: 500
Revenue: $130000.00
Procurement Cost: $65000.00
 Profit (b): $65000.00

================ PART (c): With Transshipment Service ================
Average Sales per Location:
  Jurong West: 294.91
  Orchard: 495.86
  Harbour Front: 489.66
Average Profit (c): $62728.61
Profit change vs (a): $885.53
Decision: Adopt service

================ PART (d): Optimal Inventory (SAA) ================
Using SAMPLE_SIZE_D = 50 (reduced to avoid crash)
Expected Profit (d): $63059.05
Optimal Inventory per Location:
  Jurong West: 299.44
  Orchard: 499.68
  Harbour Front: 499.05
Total Procured: 1298.17
```

###Question 5

```
[8]: # --- Parameters ---
T = [1, 2, 3, 4]
d = {1: 3, 2: 2, 3: 3, 4: 2}          # demand per season
K = {1: 2, 2: 2, 3: 2, 4: 2}          # setup cost (million)
h = {1: 0.2, 2: 0.2, 3: 0.2, 4: 0.2}# holding cost (million/unit/season)

M = 10                                # big M (>= total demand)

# --- Model ---
m = Model("Economic_Lot_Sizing")
m.setParam('OutputFlag', False)

# Decision variables
y = m.addVars(T, name="Production", lb=0)       # number produced
x = m.addVars(T, name="Inventory", lb=0)        # inventory at end of season
z = m.addVars(T, vtype=GRB.BINARY, name="Setup")   # setup binary

# --- Objective ---
m.setObjective(
    quicksum(K[t]*z[t] + h[t]*x[t] for t in T),
    GRB.MINIMIZE
)

# --- Constraints ---
# Flow balance: inventory and production must meet demand
for t in T:
    if t == 1:
        m.addConstr(y[t] - d[t] == x[t], name=f"Balance_{t}")
    else:
        m.addConstr(x[t-1] + y[t] - d[t] == x[t], name=f"Balance_{t}")

# Setup-production link
for t in T:
    m.addConstr(y[t] <= M * z[t], name=f"SetupLink_{t}")

# --- Solve ---
m.optimize()

# --- Output ---
print("\nOptimal production plan:")
for t in T:
    print(f"Season {t}: Produce {y[t].x:.0f}, Setup = {int(z[t].x)}, Ending␣
 ↪inventory = {x[t].x:.0f}")

print(f"\nTotal minimum cost: {m.objVal:.2f} million dollars")
```

```
Optimal production plan:
```

```
Season 1: Produce 10, Setup = 1, Ending inventory = 7
Season 2: Produce 0, Setup = 0, Ending inventory = 5
Season 3: Produce 0, Setup = 0, Ending inventory = 2
Season 4: Produce 0, Setup = 0, Ending inventory = 0

Total minimum cost: 4.80 million dollars
```

###Question -6

[9]:
```python
# List all optimal TSP tours (start/end at 1) without duplicates



N = [1,2,3,4,5,6]
E = [(i,j) for i in N for j in N if i != j]     # NO self-arcs
dist = {
    (1,2):3,(1,3):3,(1,4):10,(1,5):9,(1,6):10,
    (2,1):3,(2,3):3,(2,4):7,(2,5):6,(2,6):7,
    (3,1):3,(3,2):3,(3,4):7,(3,5):6,(3,6):7,
    (4,1):10,(4,2):7,(4,3):7,(4,5):1,(4,6):2,
    (5,1):9,(5,2):6,(5,3):6,(5,4):1,(5,6):1,
    (6,1):10,(6,2):7,(6,3):7,(6,4):2,(6,5):1,
}
# fill missing symmetric entries with 0 on diagonal (never used)
for i in N:
    dist.setdefault((i,i),0)

def build_model():
    m = Model("tsp_enum")
    # x only for i != j
    x = m.addVars(E, vtype=GRB.BINARY, name="x")
    # MTZ u variables, fix u1 = 0 (break symmetry)
    u = {1: m.addVar(lb=0, ub=0, name="u[1]")}
    for i in N[1:]:
        u[i] = m.addVar(lb=1, ub=len(N)-1, name=f"u[{i}]")

    # degree constraints
    for i in N:
        m.addConstr(quicksum(x[i,j] for j in N if j != i) == 1,␣
↪name=f"out[{i}]")
        m.addConstr(quicksum(x[j,i] for j in N if j != i) == 1, name=f"in[{i}]")

    # MTZ subtour elimination (for i,j != 1)
    for i in N[1:]:
        for j in N[1:]:
            if i != j:
                m.addConstr(u[i] - u[j] + (len(N)-1)*x[i,j] <= len(N)-2,
                            name=f"mtz[{i},{j}]")
    # objective
    m.setObjective(quicksum(dist[i,j]*x[i,j] for (i,j) in E), GRB.MINIMIZE)
    m.setParam("OutputFlag", 0)
    return m, x

def extract_tour(xvals):
    # follow arcs starting at 1
    tour = [1]
    cur = 1
    while True:
        nxt = next(j for j in N if j != cur and xvals.get((cur,j),0) > 0.5)
        if nxt == 1:
            tour.append(1)
            break
        tour.append(nxt)
        cur = nxt
    return tour

def canonicalize(tour, collapse_reverse=True):
    """
    Returns a canonical tuple representation to dedupe tours.
    If collapse_reverse=True, treat reverse direction as the same tour.
    """
    # remove the final 1 for canonicalization
    core = tour[:-1]
    # rotate so it always starts at 1
    assert core[0] == 1
    # forward and reverse tuples (start at 1)
    fwd = tuple(core)
    rev = tuple([1] + list(reversed(core[1:])))
    return min(fwd, rev) if collapse_reverse else fwd

# --------- solve for best cost once ----------
m, x = build_model()
m.optimize()
best = m.objVal

# Force subsequent solves to find tours with the same minimum cost
m.addConstr(quicksum(dist[i,j]*x[i,j] for (i,j) in E) <= best + 1e-6,␣
 ↪name="optcost")

tours_seen = set()
unique_tours = []

while True:
    m.optimize()
    if m.status != GRB.OPTIMAL or m.objVal > best + 1e-6:
            break

        # current tour arcs
        arcs = [(i,j) for (i,j) in E if x[i,j].X > 0.5]
        xvals = {e: x[e].X for e in E}
        tour = extract_tour(xvals)
        key = canonicalize(tour, collapse_reverse=True)  # set to False if you want␣
 ↪both orientations

        if key not in tours_seen:
            tours_seen.add(key)
            unique_tours.append(tour)
            print(f"Found optimal tour #{len(unique_tours)}: {tour}  time {best}")

        # add a no-good cut to forbid this *exact* directed tour
        m.addConstr(quicksum(x[i,j] for (i,j) in arcs) <= len(N) - 1)

print(f"\nTotal distinct optimal tours found: {len(unique_tours)}")
```

```
Found optimal tour #1: [1, 2, 5, 4, 6, 3, 1]  time 22.0
Found optimal tour #2: [1, 3, 5, 6, 4, 2, 1]  time 22.0
Found optimal tour #3: [1, 3, 4, 6, 5, 2, 1]  time 22.0
Found optimal tour #4: [1, 3, 6, 5, 4, 2, 1]  time 22.0
Found optimal tour #5: [1, 2, 6, 5, 4, 3, 1]  time 22.0
Found optimal tour #6: [1, 3, 5, 4, 6, 2, 1]  time 22.0

Total distinct optimal tours found: 6
```

[10]:
```
# ------------------------------
# data
# ------------------------------
N = [1, 2, 3, 4, 5, 6]
dist = {
    (1,1):0,(1,2):3,(1,3):3,(1,4):10,(1,5):9,(1,6):10,
    (2,1):3,(2,2):0,(2,3):3,(2,4):7,(2,5):6,(2,6):7,
    (3,1):3,(3,2):3,(3,3):0,(3,4):7,(3,5):6,(3,6):7,
    (4,1):10,(4,2):7,(4,3):7,(4,4):0,(4,5):1,(4,6):2,
    (5,1):9,(5,2):6,(5,3):6,(5,4):1,(5,5):0,(5,6):1,
    (6,1):10,(6,2):7,(6,3):7,(6,4):2,(6,5):1,(6,6):0,
}
# latest arrival (time windows)
L = {1: 1e4, 2: 5, 3: 10, 4: 15, 5: 13, 6: 14}
M = 1e4

m = Model("tsp_tw_fix")

# binary arcs (no self arcs)
```

```python
x = m.addVars([(i,j) for i in N for j in N if i != j],
              vtype=GRB.BINARY, name="x")

# arrival times
t = m.addVars(N, vtype=GRB.CONTINUOUS, name="t")

# MTZ vars (u1 = 0)
u = {1: m.addVar(lb=0, ub=0, name="u[1]")}
for i in N[1:]:
    u[i] = m.addVar(lb=1, ub=len(N)-1, name=f"u[{i}]")

# objective
m.setObjective(
    quicksum(dist[i,j]*x[i,j] for i in N for j in N if i != j),
    GRB.MINIMIZE
)

# depot start time
m.addConstr(t[1] == 0)

# in/out degree
for i in N:
    m.addConstr(quicksum(x[i,j] for j in N if j != i) == 1)
    m.addConstr(quicksum(x[j,i] for j in N if j != i) == 1)

# time propagation: ONLY to customers (j != 1)
for i in N:
    for j in N:
        if i != j and j != 1:
            m.addConstr(t[j] >= t[i] + dist[i,j] - M*(1 - x[i,j]))

# time windows (for all; 1 is loose)
for i in N:
    m.addConstr(t[i] <= L[i])

# MTZ subtour elimination (skip node 1)
for i in N[1:]:
    for j in N[1:]:
        if i != j:
            m.addConstr(u[i] - u[j] + (len(N)-1)*x[i,j] <= len(N)-2)

m.setParam("OutputFlag", 0)
m.optimize()

if m.status == GRB.OPTIMAL:
    print("Optimal total time:", m.objVal)
    # rebuild tour
    route = [1]
    cur = 1
    while True:
        nxt = [j for j in N if j != cur and x[cur,j].X > 0.5][0]
        route.append(nxt)
        if nxt == 1:
            break
        cur = nxt
    print("Route:", route)
    print("Arrival times:")
    for i in N:
        print(f"  node {i}: {t[i].X:.1f}")
else:
    print("No feasible route.")
```

```
Optimal total time: 25.0
Route: [1, 2, 3, 5, 6, 4, 1]
Arrival times:
  node 1: 0.0
  node 2: 3.0
  node 3: 6.0
  node 4: 15.0
  node 5: 12.0
  node 6: 13.0
```